Contents

ANALYSIS & DESIGN OF ALGORITHM (B.E., IV-SEM.)

UNIT 1

ALGORITHMS, DESIGNING ALGORITHMS, ANALYZING ALGORITHMS

Q.1. Define an algorithm.

Ans. Any well-defined computational procedure that takes some value, or sets of values, as input and produces some value, or sets of values, as output is called an ***algorithm***.

So, in other words, we can say that a sequence of computational steps that transforms input into the output is known as an algorithm.

Q.2. Is there any difference among algorithm, pseudocode and program ? Explain. ***(R.G.P.V., June 2017)***

Ans. Difference among algorithm, pseudocode and program are as follows –

(i) An algorithm is well defined sequenced collection of finite set of instructions for solving a particular problem.

Pseudocode is an approach of writing algorithm in the programming friendly language (exactly in programming language is not necessary).

A program is sequenced collection of instructions written in a particular language (called programming language like C, C++, Java etc.) which commands computer to perform specific task/tasks.

(ii) A pseudocode or program can be called as algorithm but vice-versa is not true.

(iii) Program can be called as pseudocode but vice-versa is not true.

(iv) Algorithms are written only is sequence, pseudocodes are written in sequence and by using some predefined keywords but program are written in sequence, using some predefined keywords and also it follows some rules govern by grammar of programming language.

(v) Algorithm can be thought of as rough idea of any product, pseudocode is blueprint of that product and program is the final product.

Q.3. Discuss important characteristics of an algorithm.

Ans. An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. Below, we have some criteria or characteristics that are to be followed by all the algorithms –

(i) Input – An algorithm should take zero or more quantities as input, that are externally supplied.

(ii) Output – An algorithm should produce at least one output.

(iii) Definiteness – Each and every instruction in an algorithm should be very clear and unambiguous.

(iv) Finiteness – An algorithm must be such that if we trace out its instructions, then for all cases, the algorithm terminates after a finite number of steps.

(v) Effectiveness – In an algorithm, every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion (iii); it must also be feasible.

For example, in recursion, if we talk about factorial function, it could be given as –

(a) if $n = 0$, then $n! = 1$

(b) if $n > 0$, then $n! = n.(n - 1)!$.

If we write algorithm for finding factorial of any number then it will follow all the above specified criteria in following manner –

(i) Input – A number n whose factorial is to be found.

(ii) Output – A number, i.e., factorial of number n.

(iii) Definiteness – Each instruction is definite, i.e., the function to be done is to just multiply the number n with one less than that of it, i.e., $n - 1$, means –

$$n(n - 1).$$

(iv) Finiteness – Algorithm will terminate when n will become zero as $0! = 1$.

(v) Effectiveness – Each instruction, i.e., $n(n - 1)$ is very basic and can be done using pencil and paper.

Q.4. How can the algorithms be designed ?

Ans. The term designing algorithm means creating an algorithm that accomplishes a particular task. Creating an algorithm is an art which may never be fully automated.

There are many ways to design algorithms. We have many design techniques that have proven to be useful in that they have often yielded good algorithms. These design strategies make it easy to devise new and useful algorithms. For example, some of design approaches are "Divide-and-Conquer", "Dynamic Programming", etc.

A common approach to solving a problem is to partition the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole.

Q.5. What do you mean by performance analysis of an algorithm ? Explain. ***(R.G.P.V., Dec. 2014)***

Ans. The efficiency of an algorithm can be decided by measuring the performance of an algorithm. The performance of an algorithm can be measured by computing two factors –

(i) Amount of time required by an algorithm to execute (time complexity)

(ii) Amount of storage required by an algorithm (space complexity).

Q.6. Explain the various criteria used for analyzing algorithm. ***(R.G.P.V., May 2018)***

Or

What criteria are used during the analysis of the algorithm ? ***(R.G.P.V., Dec. 2017)***

Ans. There are many criteria upon which we can judge algorithm, as by judging following things –

(i) Does it do what we want it to do ?

(ii) Does it work correctly according to the original specifications of task ?

(iii) Is there documentation that describes how to use it and how it works ?

(iv) Are procedure created in such a way that they perform logical subfunctions ?

(v) Is the code readable ?

The above criteria are all vitally important when it comes to writing software, most especially for large systems.

All the above criteria for judging algorithms are important, but two most important factors are –

(i) Space complexity (ii) Time complexity.

Q.7. Define algorithm. Discuss how to analyse algorithms. ***(R.G.P.V., June 2016)***

Ans. **Algorithm** – Refer to Q.1.

Analyzing an algorithm means analysis of an algorithm. Analyzing a algorithm has come to mean predicting the resources that the algorithm require

When an algorithm is executed, it acquires some of the resources suc as computer's central processing unit (CPU) to perform operations and it memory (both immediate and auxiliary) to hold the program and data communication bandwidth, or logic gates, but most often it is computationa time and storage an algorithm requires that we want to measure.

This is not easy to measure. Sometimes great mathematical skills ar required to measure it. Mathematical tools required may include discret combinatorics, elementary probability theory, algebraic dexterity, and the abilit to identify the most significant terms in a formula.

Because the behaviour of an algorithm may be different for each possibl input, we need a means for summarizing that behaviour in simple, easil understood formula.

During the analysis of an algorithm our one immediate goal is to find a means of expression that is simple to write and manipulate, shows the importan characteristics of an algorithm's resource requirements, and suppresses tediou details.

For example, the time taken by the ***insertion-sort*** procedure depends o the input. As sorting hundreds of numbers takes longer than sorting five numbers Secondly, ***insertion-sort*** can take different amounts of time to sort two inpu sequences of the same size depending on how nearly sorted they already are.

Actually, the time taken by an algorithm grows with the size of the input We describe the running time of a program as a function of the size of its input

Performance analysis of an algorithm enables us to make quantitativ judgements about the value of one algorithm over another. Also, it allows on to predict whether the software will meet any efficiency constraints that exis

Finally, analysis of an algorithm describes how well an algorithm perform in the best-case, in the worst-case and in average-case.

Q.8. How is an algorithm analysed ? What do you understand by best-case average-case and worst-case of an algorithm ? (R.G.P.V., Dec. 2011, June 2013)

Ans. **Algorithm Analysed** – Refer to Q.7.

During the analysis of an algorithm we check the algorithm for different different inputs. Depending on the time taken by an algorithm to execut different input, worst-case, average-case, and best-case are measured.

(i) Worst-case – The worst-case running time of an algorithm is a upper bound on the running time for any input. Knowing it gives us a guarante that the algorithm will never take any longer. We need not make some educate guess about the running time and hope that it never gets much worse.

For example, in insertion-sort algorithm, worst-case is one in which the input array is in reverse order. Similarly, in searching a database for a particular piece of information, the searching algorithms worst case will often occur when the information is not present in the database. And in this case of searching an absent information may be frequent.

(ii) Average-case – The average-case running time of an algorithm is an average bound on the running time for any input.

Roughly, it is often as bad as the worst-case.

For example, if we randomly choose n numbers and apply insertion-sort, we have to calculate average time to determine where in subarray A[1.....i – 1] to insert element A[i] ? In an average-case, half of the elements in A[i.....i – 1] are less than A[i] and half the elements are greater. In such a case we have to check only half of the elements of the subarray A[1.....i – 1], so

$$t_i = i/2.$$

Generally, average-case running time turns out to be a quadratic function of the input size, i.e., similar to a worst-case running time.

(iii) Best-case – The best-case running time of an algorithm is a lower bound on the running time for any input.

In the best-case we have to do minimum number of comparisons.

For example, if we take a sorted array A [1.....i – 1], we have to apply insertion sort on this array so as to insert element A[i].

If array A[1.....i – 1] is sorted in increasing order and element A[i] is smaller than A[1]in such a condition, we need to make only one comparison to find the position of element A[i] in array A[1.....1 – i], which is minimum number of comparisons in above case, so is a best-case.

Q.9. Explain how time complexity of an algorithm can be calculated. Give suitable example. ***(R.G.P.V., Dec. 2012)***

Or

How time complexity of an algorithm is calculated ? Explain with suitable example. ***(R.G.P.V., Dec. 2013)***

Ans. Time complexity of an algorithm is the amount of computer time it needs to run to completion. The time T(P) taken by a program P is the sum of compile time and run time. Also, a program compiled once can be executed many times without recompiling it.

The compile time does not depend on the instance characteristics.

Run time is denoted by t_p (instance characteristics). We should concern ourself in computing t_p.

For calculating the run time of a program we have to calculate tim taken in addition, multiplication, subtraction, division, comparison, load, store and so on. So, we could obtain an expression for $t_p(n)$ as follows –

$$t_p(n) = C_a ADD(n) + C_s SUB(n) + C_m MUL(m) + C_d DIV(n) + ...$$

Here, n denotes the instance characteristic and C_a, C_m, C_s, C_d deno time needed for an addition, multiplication, subtraction, and division.

Determining the number of addition, multiplication, subtraction a different, so we have another way to find time complexity. We count the tot number of operations. We can count the number of program steps for findi execution time which is easier than that of earlier one. The number of steps any program depends on the kind of statement.

Different statements have different counts.

For example,

(i) Comments count as zero steps.

(ii) Assignment statements are counted as one, as they do not ca any other algorithm.

(iii) For iterative statements such as for, while, and repeat-unt statements we count the steps only for control part.

For example,

for i: = <expr1> to <expr2> do

while (<expr1>) do

(a) In case of while statement, expression <expr1> has given count that is count for step.

(b) In case of for loop, we have count equal to sum of th counter for <expr1> and <expr2>.

Remaining execution for the for statement have step count of 1 and so o

When we want to know how the computing time (i.e., time complexity increases as the number of inputs increases. In this case, the number of step will be computed as a function of the number of inputs alone.

When we want to know how the computing time increases as th magnitude of one of the input increases then we have to find the number o steps as the function of the magnitude of this input alone.

So, in above way, step counting will help us in finding time complexity

Q.10. What are the factors which affect the running time of an algorithm ?
(R.G.P.V., Dec. 201

Ans. Many factors affects the running time of an algorithm or progra Some of them are listed below –

(i) Parallelism – A multi-threaded program running on multicore machi will be faster.

(ii) CPU Utilization – If CPU is already utilized by some other processes then running time of algorithm will increase.

(iii) I/O Bound – Sometimes I/O bounds like disk read/write speed affect the running time.

(iv) Overhead Due to many Function Call – There could be huge overhead on running if a function call is made multiple times for a small function.

(v) Recursion – Recursion can cause a lot of overhead which increases the running time of an algorithm.

(vi) Disk Layout – For multiple disk, RAID might work best or faster than NAS, SAN or other storage layout.

(vii) Contention – If there are multiple threads, they are synchronized to access common resources, then there can be contention for the resources which causes thread to wait.

(viii) Buggy Synchronization – If there are deadlocks, the algorithm or program will stop working or the two threads involved in deadlock will stop. This increases running time of any algorithm.

Q.11. How recursive algorithms are analyzed ? Analyze the execution time of recursive algorithm for tower of Hanoi; problem. (R.G.P.V., Dec. 2014)

Ans. The steps to be followed while analyzing recursive algorithms are as follows –

(i) Decide the input size based on parameter n.

(ii) Identify algorithm's basic operations.

(iii) Check how many times the basic operations is executed. Thereafter determine the execution of basic operation depends upon the input size n. Find worst, average and best cases for input of size n. If the basic operation depends upon worst case, average case and best case then that has to be analyzed separately.

(iv) Set up the recurrence relation with some initial condition and expressing the basic operation.

(v) Solve the recurrence or atleast determine the order of growth. While solving the recurrence we will use the forward and backward substitution method. And then correctness of formula can be proved with the help of mathematical induction method.

Towers of Hanoi – There are three pegs namely X, Y and Z. The five disks of different diameters are placed on peg X. The arrangement of the disks is such that every smaller disk is placed on the larger disk. The problem of tower of Hanoi states that move the disks from peg X to peg Z using peg Y

as an auxiliary. The conditions are –

(i) Only the top disk on any peg may be moved to any other peg.

(ii) A large size disk should never be place on smaller one.

Algorithm 1.1 Tower of Hanoi

```
Algorithm TOH(n, X, Y, Z)
{
if (n = 1) then                    //if one disk has to be moved
{
    write("The peg moved from X to Z")
    return
}
else
{
TOH(n – 1, X, Y, Z); //move top n – 1 disks from X to Y using Z
TOH(n – 1, Y, Z, X); //move remaining disk from Y to Z using X
}
}
```

The tower of Hanoi has a time complexity as

$O(2^n - 1) = O(2^n)$.

Q.12. Write an algorithm for calculation the fn Fibonacci number in Θ (log n) time. ***(R.G.P.V., June 2012)***

Ans. The algorithm to compute the fn Fibonacci number is as follows –

Algorithm 1.2 Fibonacci Number

```
Algorithm Fibonacci (fn)
{
    if   (fn ≤ 1) then
         write (fn);
    else
    {
         fn1 = 1;
         fn2 = 0;
         for i = 2 to n do
         {
              fn = fn1 + fn2;
              fn2 = fn1;
              fn1 = fn;
         }
         write (fn);
    }
}
```


ASYMPTOTIC NOTATIONS

Q.13. What do you understand by the term asymptotics ?

Or

What is the significance of asymptotic notations ? (R.G.P.V., Dec. 2017)

Ans. The word asymptotics means the study of functions of a parameter n, as n becomes larger and larger without bound. Asymptotic efficiency of algorithms is concerned when the input sizes are large enough to make only the order of growth of the running time relevant. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. For all very small inputs, algorithm that is asymptotically more efficient will be the best choice.

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mu = \{0, 1, 2, \ldots\ldots 3\}$.

Such notations are convenient for describing the worst-case running-time function T(n), which usually defines only an integer input sizes.

We can abuse asymptotic notation in variety of ways. For example, the notation is easily extended to the domain of real numbers or, alternatively, restricted to a subset of the natural numbers.

Q.14. What is the use of asymptotic notations ? (R.G.P.V., June 2015)

Ans. Asymptotic notations are used to describe the limiting behaviour of a function when the argument tends towards a particular value, usually in terms of simpler functions. Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. Various asymptotic notations are mostly used for calculating the performance or running time of an algorithm.

Q.15. What is asymptotic notations ? Explain each. (R.G.P.V., June 2011)

Or

Describe about various types of asymptotic notations used in algorithm analysis. ***(R.G.P.V., Dec. 2011)***

Or

What is the significance of various asymptotic notations ? Explain by giving examples. ***(R.G.P.V., Dec. 2012)***

Or

What are the different asymptotic notations used ? Explain.

(R.G.P.V., June 2014, Dec. 2014)

Or

What is an asymptotic notations ? Give the different notations to used to represent the complexity of algorithms. ***(R.G.P.V., May 2019)***

Ans. Refer to Q.13.

Asymptotic notations are (O, Ω, Θ).

(i) Big oh (O) Notation – The function f(n) = O(g(n)) (read as "f of n is big oh of g of n") iff (if and only if) there exists positive constant C and n_0 such that –

$f(n) \leq C * g(n)$, for all n, $n \geq n_0$.

We use O-notation to give an upper bound on a function, to within a constant factor.

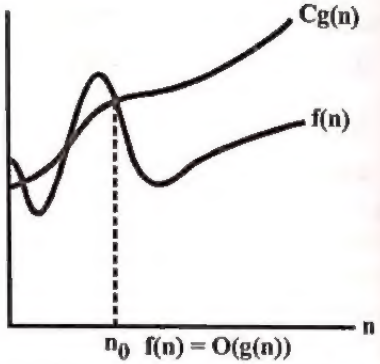


Fig. 1.1 Intuition Behind O-Notation

For example –

(a) The function $5n + 4 = O(n)$ as $5n + 4 \leq 6n$, for all $n \geq 4$.

(b) $6n + 6 = O(n)$ as $6n + 6 \leq 7n$, for all $n \geq 6$.

(c) $200n + 5 = O(n)$ as $200n + 5 \leq 201n$, for all $n \geq 5$.

(d) $10n^2 + 4n + 2 \leq 11n^2$, for all $n \geq 5$.

(e) $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$, for all $n \geq 100$.

(f) $6*2^n + n^2 = O(2^n)$ as $6 * 2^n + n^2 < 7 * 2^n$, for $n \geq 4$.

(g) $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$, for $n \geq 2$.

(h) $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ or $n \geq 2$.

(i) $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to C for any constant C and all $n \geq n_0$.

(j) $10n^2 + 4n + 2 \neq O(n)$.

Note – When we write f(n) = O(g(n)), we are merely claiming that some constant multiple of g(n) is an asymptotic upper bound on f(n), with no claim about how tight an upper bound is.

For all values of n to the right of n_0, the value of the function f(n) is on or below.

The big oh notation has different meanings depending on different values of variable n. Some of them can be stated as follows –

(a) O(1) – Indicates a constant computing time.

(b) O(n) – Indicates a linear computing time.

(c) $O(n^2)$ – Indicates a quadratic computing time.

(d) $O(n^3)$ – Indicates a cubic computing time.

(e) $O(2^n)$ – Indicates an exponential computing time.

(f) O(log n) – Indicates a logarithmic computing time.

(ii) Omega (Ω) Notation – The function f(n) = Ω (g(n)) (read as "f of n is omega of g of n") iff there exists positive constant C and n_0 such that –

$f(n) \geq C*g(n)$, for all n, $n \geq n_0$.

Just as O-notation provides an asymptotic upper bound on a function, Ω-notation provides an ***asymptotic lower bound***.

For a function g(n), we denote by Ω(g(n)) the set of functions.

$\Omega(g(n)) = \{f(n)$: there exists positive constants C and n_0 such that $0 \leq Cg(n) \leq f(n)$ for all $n \geq n_0\}$.

For example, the function $3n + 2 = \Omega(n)$, as $3n + 2 \geq 3n$, for $n \geq 1$.

(The inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$).

$3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$, for $n \geq 1$. $100n + 6 = \Omega(n)$ as $100n + 6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$, for $n \geq 1$.

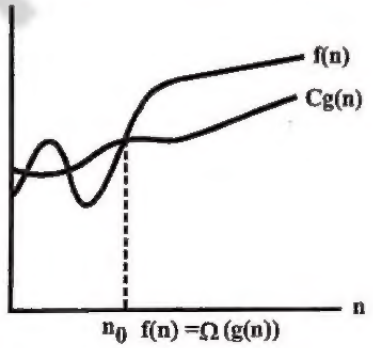


Fig. 1.2

The intuition behind Ω notation is shown in fig. 1.2.

For all values of n to the right of n_0, the value of f(n) is on or above g(n).

(iii) Theta (Θ) Notation – The function of f(n) = Θ (g(n)) (read as "f of n is theta of g of n") iff there exists positive constants C_1, C_2, and n_0, such that $C_1g(n) \leq f(n) \leq C_2g(n)$.

A function f(n) belongs to the set Θ(g(n)) if there exists positive constants C_1 and C_2 such that it can be "sandwiched" between $C_1g(n)$ and $C_2g(n)$, for sufficiently large n.

Although Θ(g(n)) is a set, we write "f(n) = Θ(g(n))".

Fig. 1.3 gives intuitive picture of function f(n) and g(n), where f(n) = Θ(g(n)).

For all values of n to the right of n_0, the value of f(n) lies at or above $C_1g(n)$ and at or below $C_2g(n)$.

In other words, for all $n \geq n_0$, the function f(n) is equal to g(n) to within a constant factor.

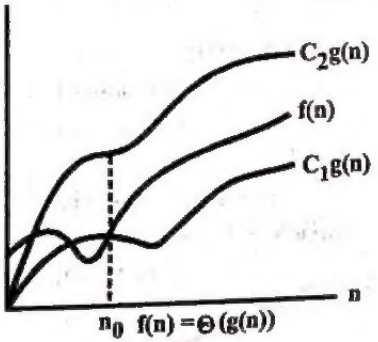


Fig. 1.3

We say that, g(n) is an asymptotically tight bound for f(n).

The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $C_1 = 3$, $C_2 = 4$, and $n_0 = 2$.

$$3n + 3 = \Theta(n),\ 10^2 + 4n + 2 = \Theta(n^2),$$

$$6 * 2^n + n^2 = \Theta(2^n), \text{ and}$$

$$10 * \log n + 4 = \Theta(\log n)$$

So, theta notation is more precise than both the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound of $f(n)$.

Q.16. Give the definition of Big "oh" with example. ***(R.G.P.V., June 2016)***

Ans. Refer to Q.15 (i).

Q.17. Define Ω notation. ***(R.G.P.V., June 2015)***

Ans. Refer to Q.15 (ii).

Q.18. What are the differences between big-oh (O), omega (ω) and theta (θ) notations ? ***(R.G.P.V., June 2017)***

Ans. Refer to Q.15.

Q.19. Why do we use asymptotic notations in the study of algorithms ? Briefly describe the commonly used asymptotic notation. (R.G.P.V., Dec. 2016)

Ans. Refer to Q.14 and Q.15.

Q.20. Briefly describe the asymptotic notations used in the study of algorithms. Also write the relational properties of the asymptotic notations. ***(R.G.P.V., Nov. 2018)***

Ans. Refer to Q.15.

The relational properties of real numbers apply to asymptotic comparisons. Suppose that $f(n)$ and $g(n)$ are asympotically positive.

Transitivity –

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,

$f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,

$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

Reflexivity –

$f(n) = \Theta(f(n))$,

$f(n) = O(f(n))$,

$f(n) = \Omega(f(n))$.

Symmetry –

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose Symmetry –

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Because these properties hold for asymptotic notations, one can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b –

$f(n) = O(g(n)) \approx a \leq b$,

$f(n) = \Omega(g(n)) \approx a \geq b$,

$f(n) = \Theta(g(n)) \approx a = b$,

$f(n) = o(g(n)) \approx a < b$,

$f(n) = \omega(g(n)) \approx a > b$.

Q.21. Define [little "oh"] "o" notation.

Ans. The asymptotic upper bound provided by o-notation may or may not be asymptotically tight. The bound $2n^2 = o(n^2)$ is asymptotically tight, but the bound $2n = o(n^2)$ is not.

We use o-notation to denote the upper bound that is not asymptotically tight.

We formally define $o(g(n))$ ("little-oh of g of n") as the set.

$o(g(n)) = \{f(n)$: for any positive constant $C > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < C\ g(n)$ for all $n \geq n_0\}$.

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The function $f(n) = o(g(n))$ iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Example – The function $3n + 2 = o(n^2)$

Since $$\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$$

$$3n + 2 = o(n \log n)$$

$$3n + 2 = o(n \log \log n)$$

$$6 * 2^n + n^2 = o(3^n).$$

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq Cg(n)$ holds for some constant $C > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < Cg(n)$ holds, for all constant $C > 0$.

Intuitively, in the o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity –

that is, $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$

Q.22. Define [little omega] "ω" notation.

Ans. By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. It can be defined as –

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, we define ω(g(n)) ("little-omega of g of n") as the set.

$\omega(g(n)) = \{f(n)$: for any positive constant $C > 0$, there exists a constant $n_0 > 0$ such that $0 \leq Cg(n) < f(n)$ for all $n \geq n_0\}$.

For example, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$.

The relation $f(n) = \omega(g(n))$ implies that

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$ if the limit exists.

That is, f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

Q.23. Write in brief about the significance of the following notations –
(i) 'O' (ii) 'Ω' (iii) 'θ' (iv) 'o' (v) 'ω'

(R.G.P.V., Dec. 2013)

Ans. Refer to Q.15, Q.21 and Q.22.

Q.24. Explain the variation of the rate of growth of common computing functions.

Ans. Time complexity of an algorithm is generally some function of the instance characteristics. This function is very important in determining how the time requirements vary as the instance characteristics change. Also, the complexity function can be used to compare two algorithms P and Q that perform the same task. Now suppose that algorithm P has complexity Θ(n) and algorithm Q has complexity $\Theta(n^2)$. Here we can assert that algorithm P is faster than algorithm Q for sufficiently large n. To confirm the validity of this assertion, observe that the computing time of P is bounded from above by ***cn*** for some constant c and for all ***n, n ≥ n₁***, whereas that of Q is bounded from below by ***dn²*** for some constant d and all n, ***n ≥ n₂***. As ***cn ≤ dn²*** for ***n ≥ c/d***, algorithm P is faster than algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

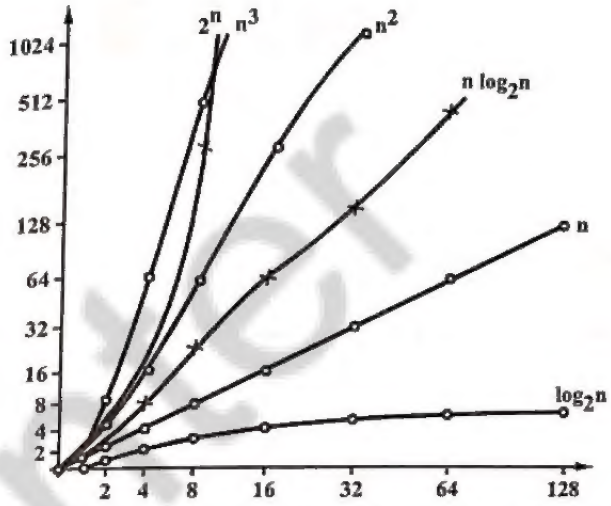


Fig. 1.4 Rate of Growth of Common Computing Time Functions

Figs. 1.4 and 1.5 describe how the computing times (counts) grow with a constant equal to one. Here it should be noticed that how the times O(n) and O (n log n) grow much more slowly than the others. For large data sets, algorithms with a complexity greater than O (n log n) are often impractical. An algorithm which is exponential will work for only small inputs. Even for exponential algorithms, if we improve the constant, say by 1/2 or 1/3, we will not improve the amount of data, we can handle, very much.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4,294,967,296

Fig. 1.5 Values for Computing Functions

Now, an algorithm is given. We analyze the frequency count of each statement and total the sum. This may give a polynomial

$$P(n) = c_k n^k + c_{k-1} n^{k-1} + \ldots\ldots + c_1 n + c_0$$

where the c_i are constants, $c_k \neq 0$ and n is a parameter. Using big-oh notation, $P(n) = O(n^k)$. On the other hand, if any step is executed 2^n times or more the expression

$$c2^n + P(n) = O(2^n)$$

Another important performance measure of an algorithm is the space it requires. Generally one can trade space for time, getting a faster algorithm but using more space.

NUMERICAL PROBLEMS

Prob.1. Find the O-notation for the function –

$$f(n) = 5n^3 + n^2 + 3n + 2$$ ***(R.G.P.V., June 2011)***

Sol. The function $5n^3 + n^2 + 3n + 2 = O(n^3)$ as $5n^3 + n^2 + 3n + 2 \leq 6n^3$ for all $n \geq 3$.

HEAP AND HEAPSORT

Q.25. Define heap. ***(R.G.P.V., Dec. 2011)***

Ans. A max (min) heap is a complete binary tree with the property that the value at each node is at least as large as (as small as) that values at its children (if they exist).We call this property ***heap property.*** Many operations can be performed on a heap.

Q.26. Give all the steps to convert the following complete binary tree into heap (max). ***(R.G.P.V., May/June 2006)***

Ans. The procedure of heapify is as follows – Create a heap out of the 8 elements. The sequential steps are given below –

(i) The initial tree is shown in fig. 1.6, since n = 8, the first step to adjust has i = 3 in fig. 1.7 (a).

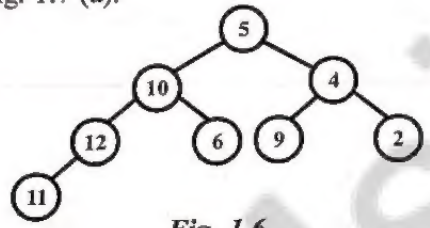


Fig. 1.6

(ii) The tree elements 4, 5 and 9 are again arranged to form a heap in fig. 1.7 (b).

(iii) Again we see that i = 1 is smaller, than i = 3, then interchange i = 1 and i = 3 as in fig. 1.7 (c).

(iv) Now in left subtree interchange i = 1 to i = 2 as shown in fig. 1.7 (d).

(v) As shown in fig. 1.7 (d) i = 2 is smaller than i = 4 (12) than change the position to each other.

(vi) We get fig. 1.7 (e). Again applying same rules we get fig. 1.7 (f).

(vii) Now in fig. 1.7 (e), only two elements are in left subtree that is not in correct order according to max heap. So rearrange them.

(viii) At last, we have a complete heap tree shown in fig. 1.7 (g).

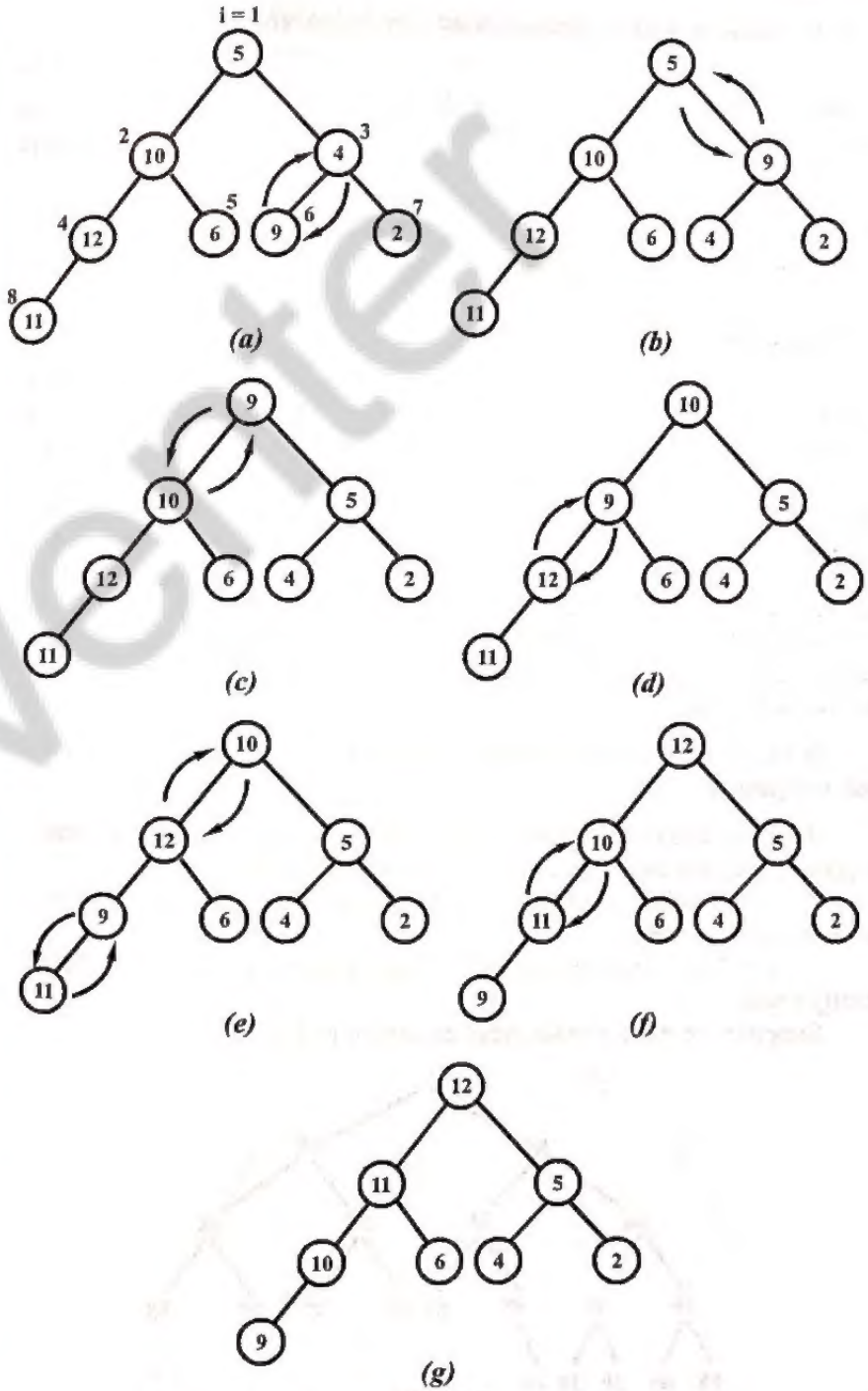


Fig. 1.7

Q.27. Show that an n-element heap has height [log n].

(R.G.P.V., June 2010)

Ans. In order to show this let the height of the n-element heap be h. From the bounds obtained on maximum and minimum number of elements in a heap, we get

$$2^h \le n \le 2^{h+1} - 1$$

where n is the number of elements in a heap.

$$2^h \le n \le 2^{h+1}$$

Taking logarithms to the base 2

$$h \le \log_2 n \le h + 1$$

It follows $h = \lfloor \log_2 n \rfloor$.

We known from above that largest element resides in root, A[1]. The natural question to ask is where in a heap might the samllest element resides ? Consider any path from root of the tree to a leaf. Because of the heap property, as we follow that path, the elements are either decreasing or staying the same. If it happens to be the case that all elements in the heap are distinct, then the above implies that the smallest is in a leaf of the tree. It could also be that an entire subtree of the heap is the smallest element or indeed that there is only one element in the heap, which in the smallest element, so the smallest element is everywhere. Note that anything below the smallest element must equal the smallest element, so in general, only entire subtrees of the heap can contain the smallest element.

Q.28. How insertion is done in a heap ? Explain and give its algorithm and complexity.

Ans. Suppose that H is a heap with n elements, and an item of information is given.Then, this insertion into heap can be done as follows –

(i) Adjoin item at the end of H so that H is still a complete tree, but not necessarily a heap.

(ii) Then, let item rise to its "appropriate place" in H so that H is finally a heap.

Suppose we have a given heap as shown in fig. 1.8.

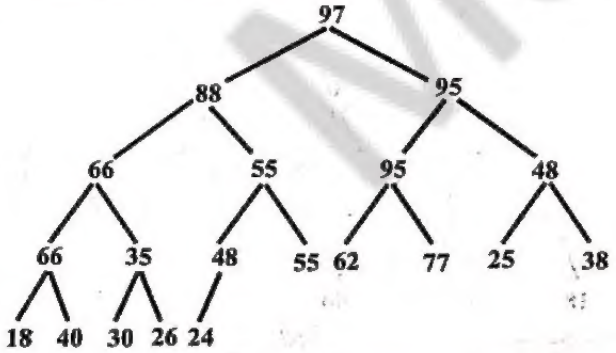


Fig. 1.8 Heap

The sequential representation of above heap is shown in fig. 1.9.

97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Fig. 1.9 Sequential Representation

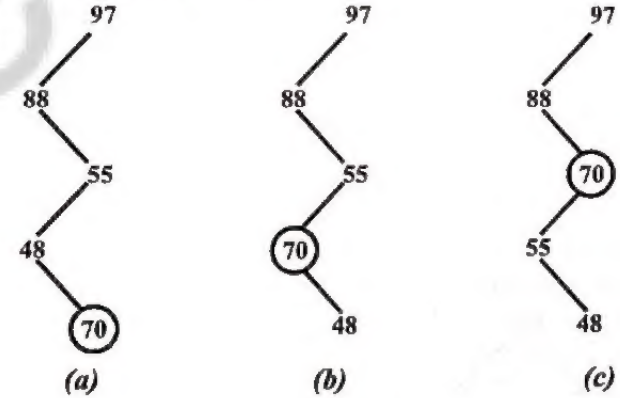
Now we add item 70 to above heap. For this we add 70 at next element position only, i.e., at position 21. Now, the heap will look like fig. 1.10 (a). We have to find the appropriate place of 70 in heap, for this we do –

(i) Compare 70 with its parent element, i.e., 48 in our case. If parent element is less than 70 then interchange the position. Otherwise we have come to know the right position.

Here, 48 < 70, so interchange the position as shown in fig. 1.10 (b).

(ii) Now, compare 70 with its new parent 55. Since 70 is greater than 55, interchange 70 and 55 the path will look like fig. 1.10 (c).

(iii) Compare 70 with 88. Since 70 does not exceed 88, item = 70 has risen to its appropriate place in H.



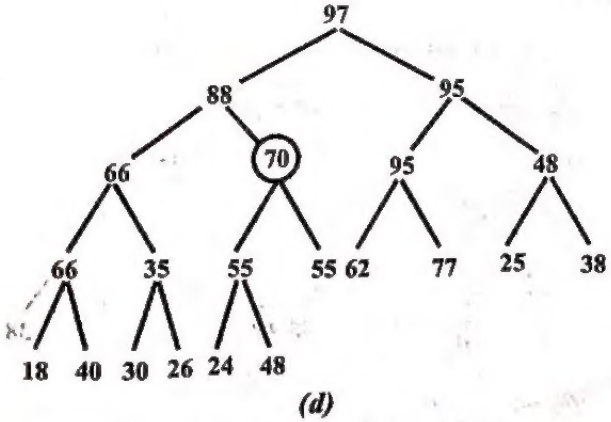


Fig. 1.10 Insertion into a Heap

The algorithm for insertion into a heap can be given as follows –

Algorithm 1.3 Insertion into a Heap

```
1   Algorithm Insert(a, n)
2   {
3       // Inserts a[n] into the heap which is stored in a[1:n – 1].
4       i := n; item := a[n];
5       while((i > 1) and (a [⌊i/2⌋] < item)) do
6       {
7               a[i] := a[⌊i/2⌋]; i := ⌊i/2⌋;
8       }
9       a[i] := item; return true;
10      }
```

Complexity – If there are n elements in the heap inserting a new element takes θ(log n) time in the worst case.

Q.29. Explain how deletion can be performed from a heap.

Ans. To delete the maximum key from the heap, we use an algorithm called adjust.

Suppose H is a heap, with N elements, and we want to delete the root R of H. This can be done as follows –

(i) Assign the root R to some variable item.

(ii) Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.

(iii) (Reheap) L sink to its appropriate place in H so that H is finally a heap.

Deletion can be understood by following example.

Let we have to delete root node 95 from heap given in fig. 1.11.

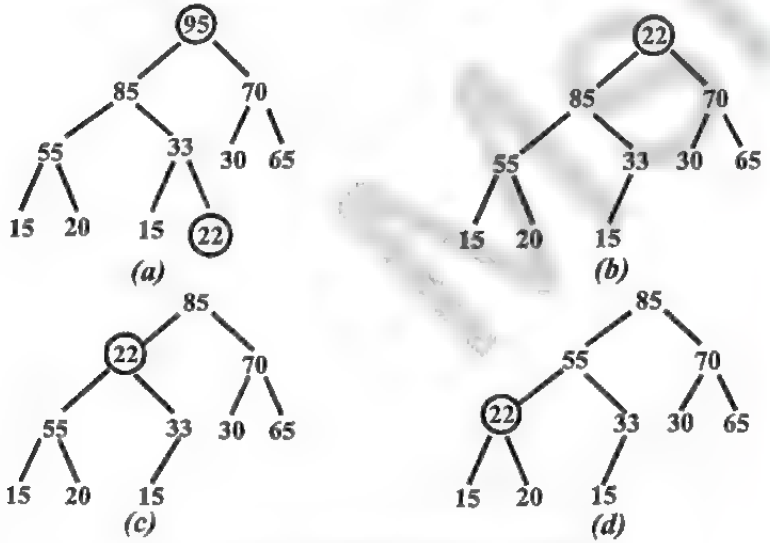


Fig. 1.11 Deletion from a Heap

We delete node 95 from heap and put last node in its place, i.e., place 22 in place of 95.

Now we have a complete tree but not a heap. For this we have to heapify it. This can be done by following steps –

(i) Compare 22 with its children and interchange its position with largest child greater than 22. So, we replace 22 and 85.

(ii) Again repeat above step with 22, and replace 22 with 55.

(iii) Again we do the same, but this time 22 is greater than both of its children, so no conversion is required.

Complexity – If there are n elements in a heap, deleting the maximum can be done in O(log n) time.

Q.30. Explain heapsort algorithm with example.
(R.G.P.V., June 2009, Dec. 2011, June 2013)

Ans. Suppose an array A with N elements is given. The heapsort algorithm to sort A consists of the two following phases –

Phase A – Build a heap H out of the elements of A.

Phase B – Repeatedly delete the root element of H.

Heapsort algorithm can be given as follows –

Algorithm 1.4 Heapsort Algorithm

```
1   Algorithm HeapSort(a,n)
2   // a[1:n] contains n elements to be sorted. HeapSort
3   // rearranges them inplace into nondecreasing order.
4   {
5       Heapify (a, n); // Transform the array into a heap.
6   // Interchange the new maximum with the element
7   // at the end of the array.
8   for i := n to 2 step – 1 do
9   {
10      t := a[i]; a[i] := a[1]; a[1] := t;
11      Adjust (a, 1, i – 1);
12  }
13  }
```

Example – Refer to Prob.4.

Q.31. What is the time complexity of heapsort ? Justify your answer.
(R.G.P.V., May/June 2006)

Ans. Heapsort requires time O (n log n). A sorting algorithm incorporates the fact that n elements can be inserted in O (n) time. Though the call of Heapify requires only O(n) operation. Adjust requires O (log n) operations for each invocation.

Thus, the worst time is O (n log n).

NUMERICAL PROBLEMS

Prob.2. Following nodes are inserted in empty tree, to form minimum heap. With neat sketches show how insertion will be done.

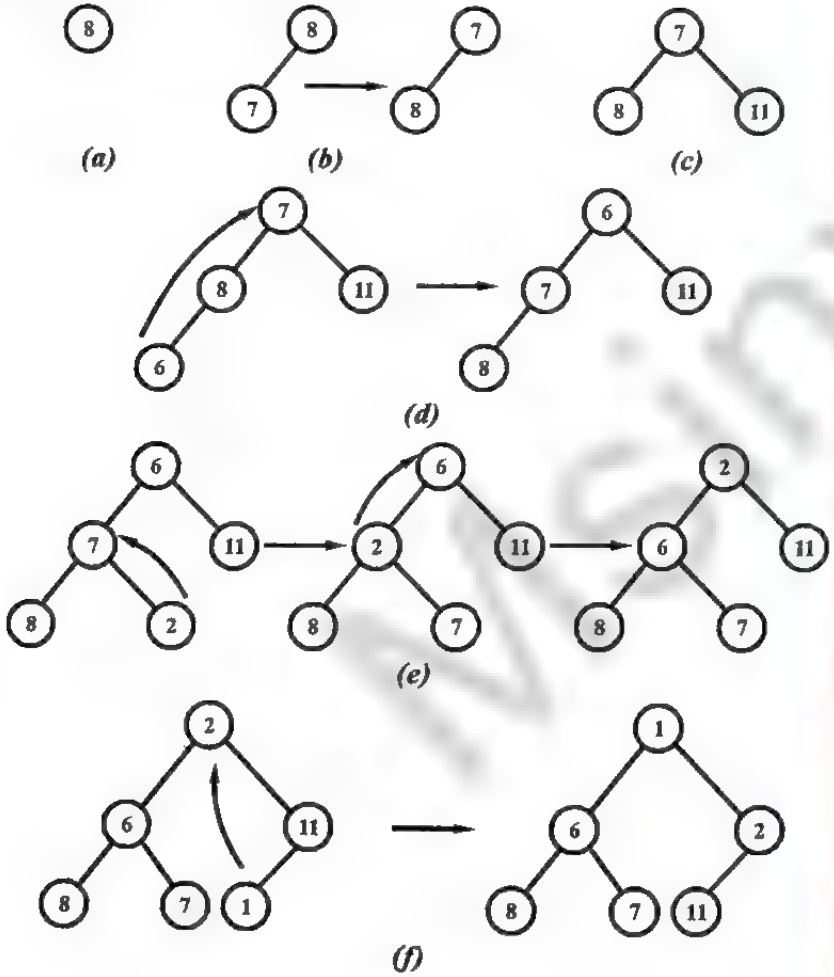
8, 7, 11, 6, 2, 1, 5, 12.

(R.G.P.V., May/June 2006, June 2014)

Sol. The given data array is

i =	1	2	3	4	5	6	7	8
A	8	7	11	6	2	1	5	12

We have 8 elements to insert in empty tree, fig. 1.12 shows how the data move around until a heap is created.



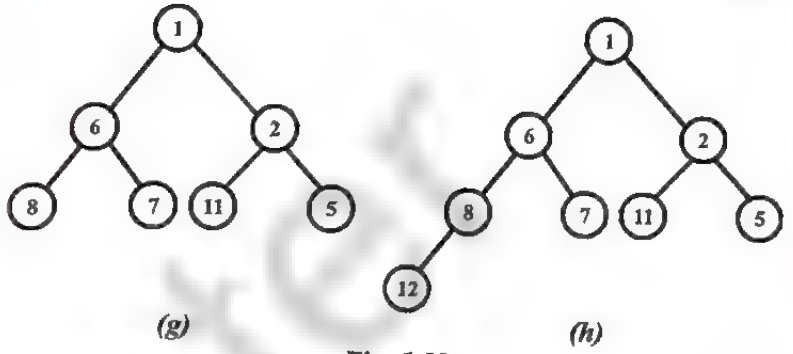


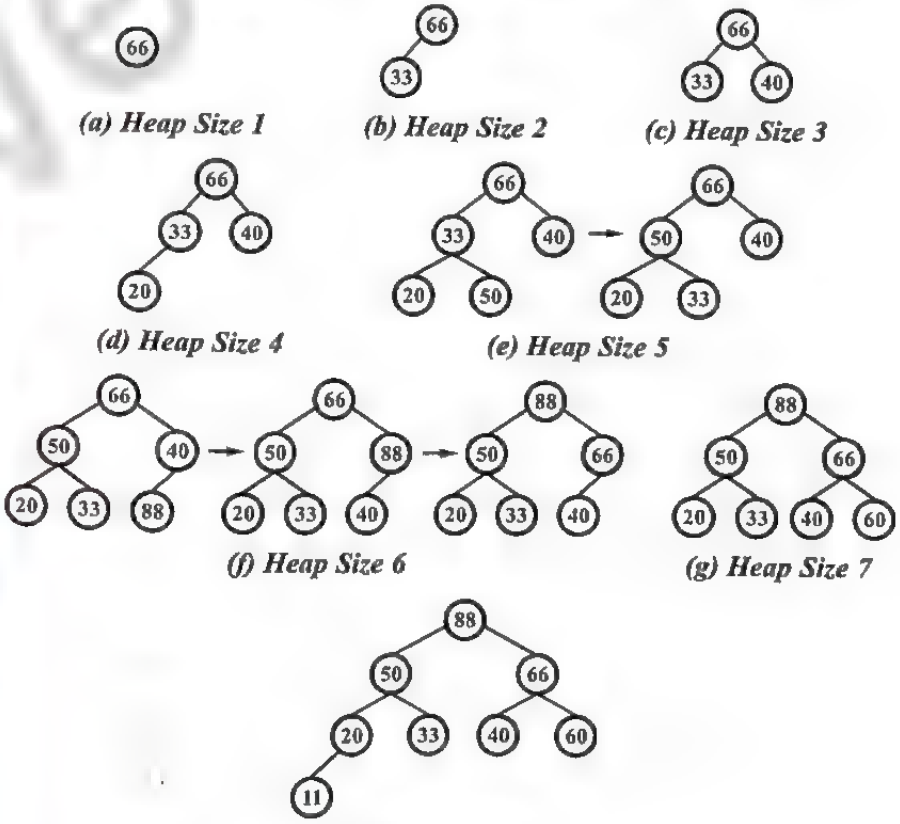
Fig. 1.12

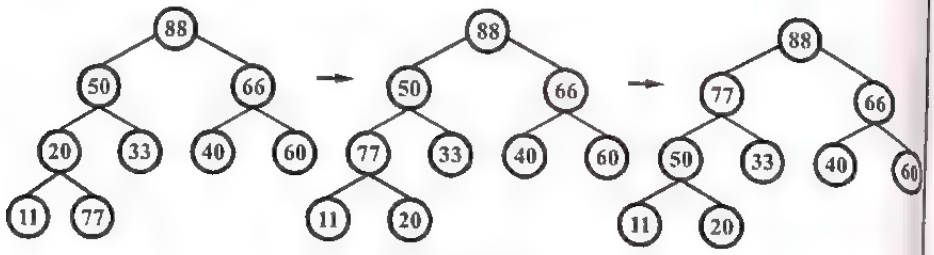
Prob.3. Sort the following list using heapsort –

66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65

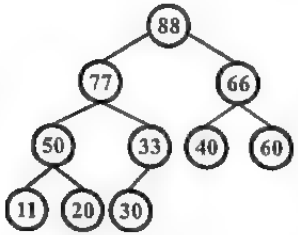
Also discuss the complexity of the heapsort. ***(R.G.P.V., Dec. 2016)***

Sol. Fig. 1.13 shows the creation of heap for the given list of keys –

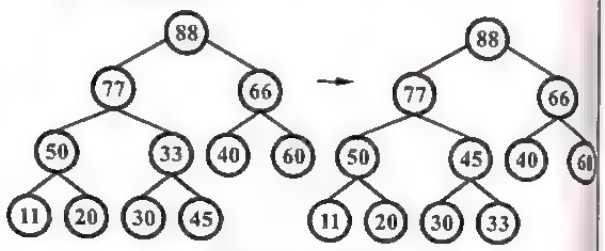




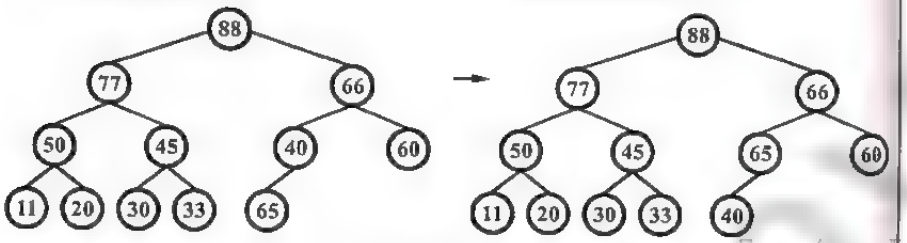
(i) Heap Size 9



(j) Heap Size 10



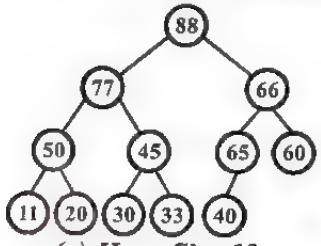
(k) Heap Size 11



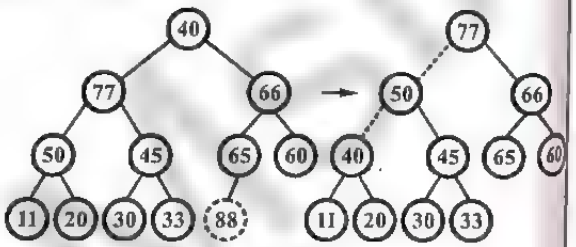
(l) Heap Size 12

Fig. 1.13 Creation of Heap

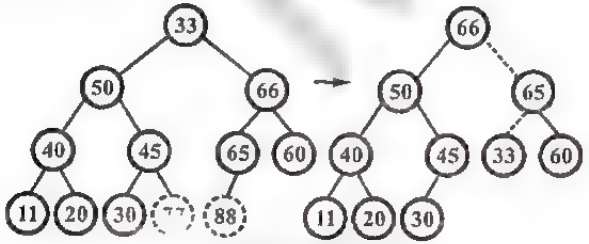
Fig. 1.14 shows processing of heap.



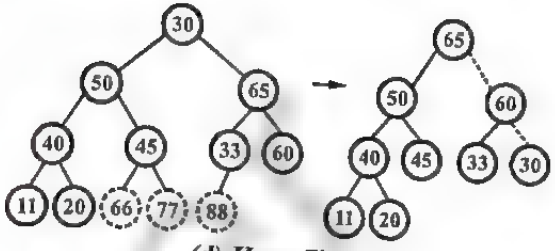
(a) Heap Size 12



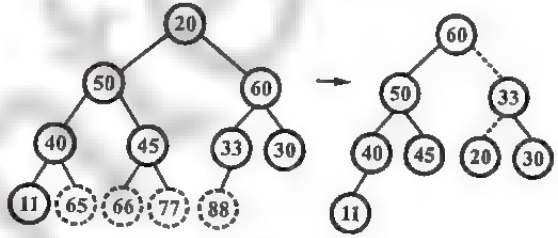
(b) Heap Size 11



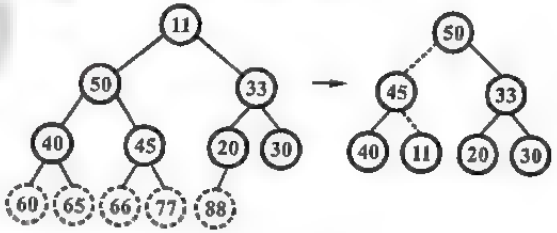
) Heap Size 10



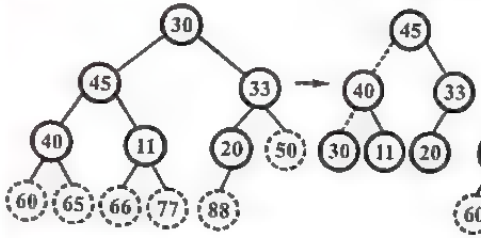
(d) Heap Size 9



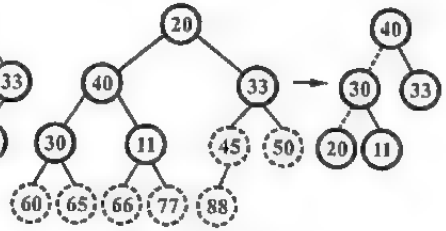
(e) Heap Size 8



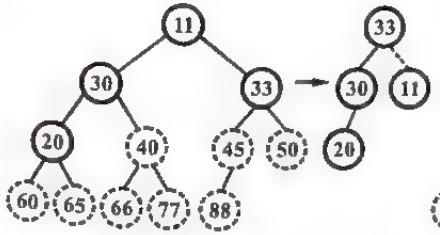
(f) Heap Size 7



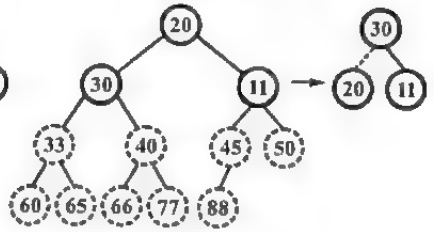
(g) Heap Size 6



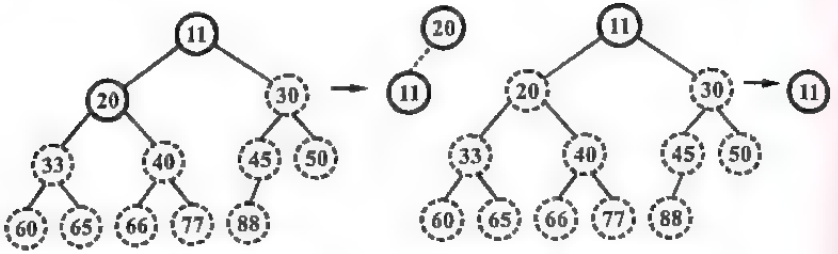
(h) Heap Size 5



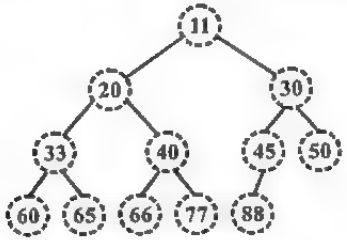
(i) Heap Size 4



(j) Heap Size 3



(k) Heap Size 2 *(l) Heap Size 1*

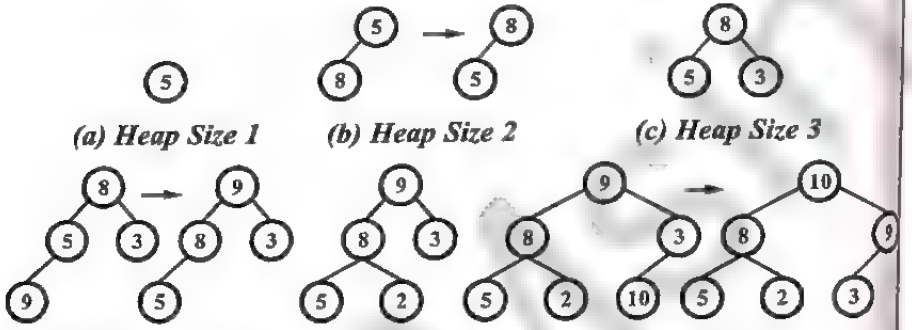


(m) Sorted Keys

Fig. 1.14

Prob.4. Illustrate heapsort on the array –
A = [5, 8, 3, 9, 2, 10, 1, 40] ***(R.G.P.V., Dec. 2015)***

Sol. Fig. 1.15 shows the creation of heap for the given list of keys –



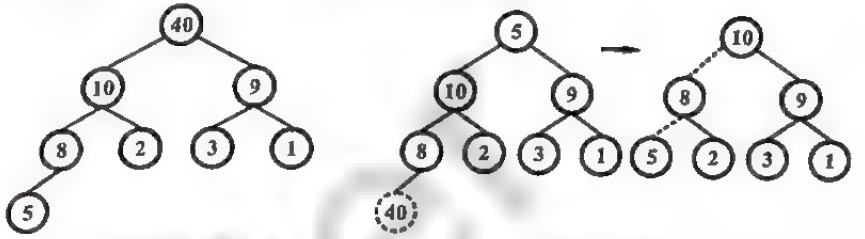
(a) Heap Size 1 *(b) Heap Size 2* *(c) Heap Size 3*

(d) Heap Size 4 *(e) Heap Size 5* *(f) Heap Size 6*

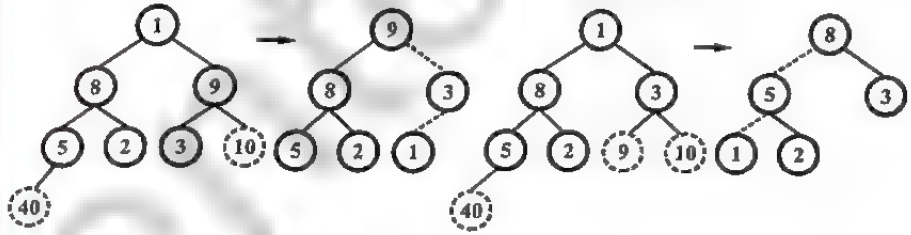
(g) Heap Size 7 *(h) Heap Size 8*

Fig. 1.15 Creation of Heap

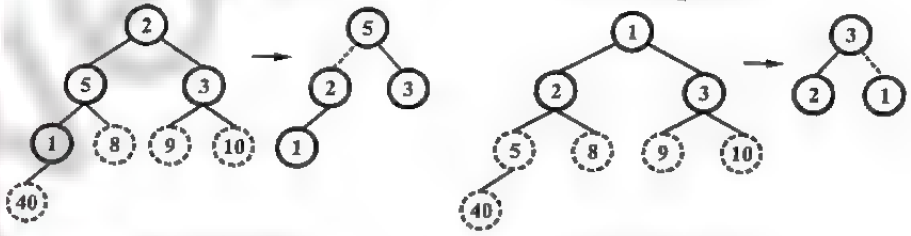
Fig. 1.16 shows the processing of heap.



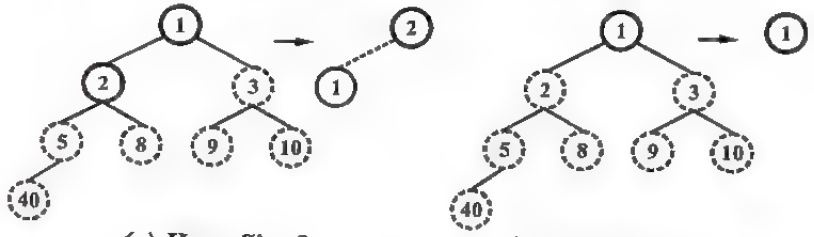
(a) Heap Size 8 *(b) Heap Size 7*



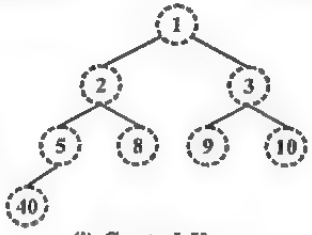
(c) Heap Size 6 *(d) Heap Size 5*



(e) Heap Size 4 *(f) Heap Size 3*



(g) Heap Size 2 *(h) Heap Size 1*



(i) Sorted Keys

Fig. 1.16

Prob.5. Illustrate the operation of heapsort on the array –

$$A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$$

(R.G.P.V., Nov. 2018

Sol. Fig. 1.17 shows the creation of heap for the given list of keys –

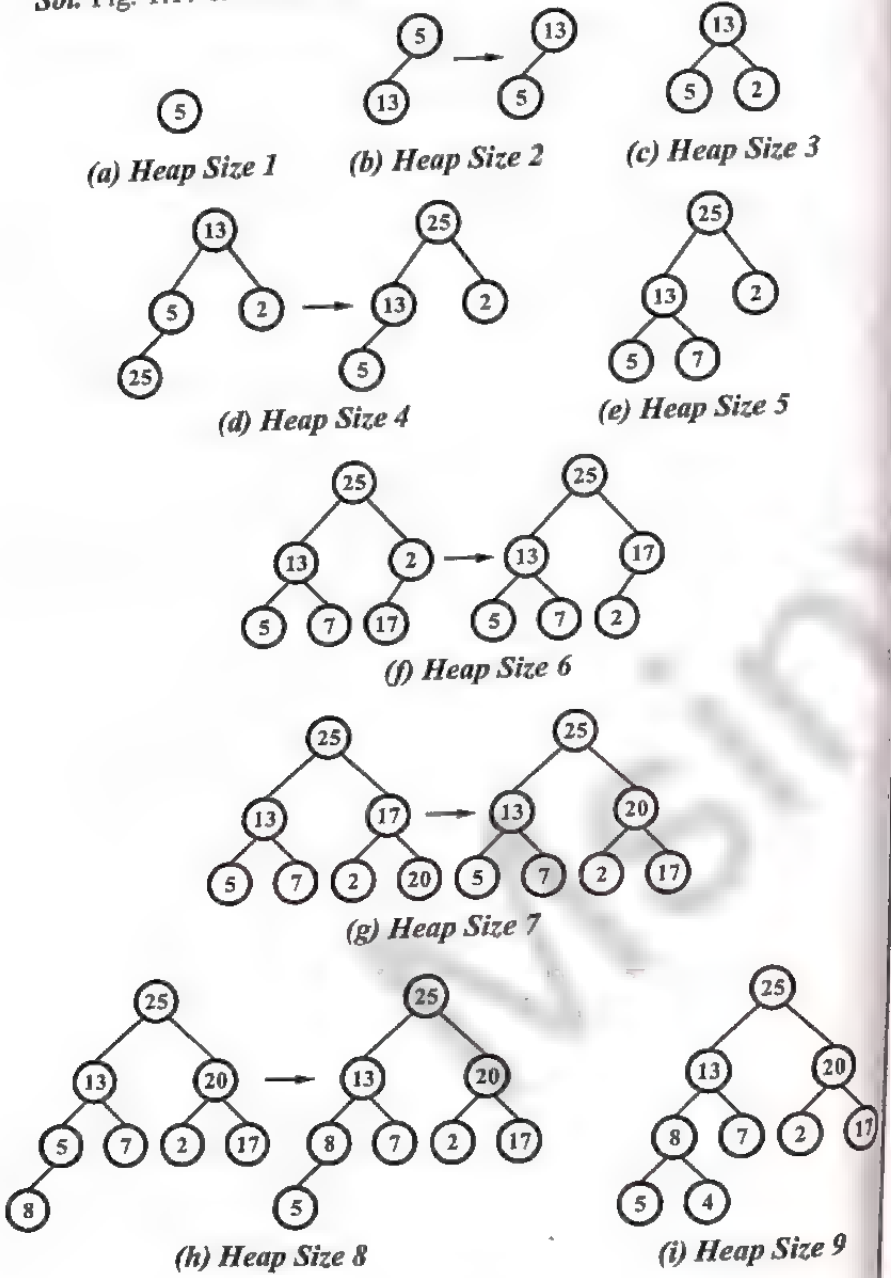


Fig. 1.17 Creation of Heap

INTRODUCTION TO DIVIDE AND CONQUER TECHNIQUE, ANALYSIS, DESIGN AND COMPARISON OF VARIOUS ALGORITHMS BASED ON THIS TECHNIQUE, EXAMPLE BINARY SEARCH, MERGE SORT, QUICK SORT, STRASSEN'S MATRIX MULTIPLICATION

Q.32. What is divide and conquer strategy ? Write and explain the algorithm of divide-and-conquer. ***(R.G.P.V., Dec. 2012)***

Ans. Consider a problem P associated with a set S. Suppose A is an algorithm which partitions S into smaller sets such that the solution of the problem P for S is reduced to the solution of P for one or more of the smaller sets. Then A is called a divide-and-conquer algorithm.

In other words, divide-and-conquer strategy suggests splitting the input into k distinct subsets, $1 < k \leq n$, yielding k subproblems. These subproblems must be solved separately and, then a method must be found to combine subsolutions into a solution of the whole.

The divide and conquer paradigm involves three steps at each level of the recursion.

Divide – Divide the problem into number of subproblems.

Conquer – Conquer the subproblems by solving them recursively. If the subproblems sizes are small enough, just solve the subproblems in a straightforward manner.

Combine – Combine the solutions of the subproblems into the solution for the original problem.

In cases where main problem and subproblems are of same type, then divide-and-conquer principle is naturally expressed by a recursive algorithm.

Three examples of the divide and conquer algorithms are –

(i) Binary search algorithm

(ii) Mergesort algorithm

(iii) Quicksort algorithm.

Algorithm – By a control abstraction we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. DAndC (algorithm 1.5) is initially invoked as DAndC(P), where P is the problem to be solved.

Small(P) is a boolean-valued function that determines whether the input size is small enough if so, then the answer can be computed without splitting.

If its value is true means problem is small enough and can be easily solved. If its value is false means problem is not small enough and it should be partitioned into smaller problems.

These subproblems $P_1, P_2,....,P_k$ are solved by recursive applications o DAndC.

At the end, Combine is a function that determines the solution to P usin the solutions to the k subproblems.

If the size of problem P is l, and the sizes of k subproblems are l_1, l_2,.....,l_k, respectively, then the computing time of DAndC is described b the recurrence relation –

$$T(l) = \begin{cases} g(l) & l \text{ small} \\ T(l_1) + T(l_2) + \ldots\ldots\ldots\ldots + T(l_k) + f(l) & \text{otherwise.} \end{cases}$$

Here, $g(l)$ is the time to compute answer directly.

$F(l)$ is the time for dividing P and combining the solutions to subproblem

$T(l)$ is the time for DAndC on any input of size l.

The complexity of many divide-and-conquer algorithms is given b recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Here, a and b are constants.

Here, we consider that value of T(1) is known, and n is defined as pow of b (i.e., $n = b^k$).

This complexity can be found by solving above recurrence relation. O of the method is ***substitution.***

Algorithm 1.5 Divide-and-conquer Algorithm

```
1.  Algorithm DAndC(P)
2.  {
3.     if Small(P) then return S(P);
4.     else
5.     {
6.        divide P into smaller instances P1, P2,....,Pk, k ≥ 1;
7.        Apply DAndC to each of these subproblems;
8.        return Combine (DAndC(P1), DAndC(P2),...., DAndC(Pk))
9.     }
10. }
```

Q.33. Explain the divide and conquer technique. Write an algorithm find the maximum and minimum element in an array using this techniq ***(R.G.P.V., Dec. 20***

Ans. **Divide and Conquer Technique** – Refer to Q.32.

Algorithm 1.6 Recursively Finding the Maximum and Minimum

```
1.  Algorithm MaxMin(i, j, max, min)
2.  // a[1 : n] is a global array. Parameters i and j are integers,
3.  // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4.  // largest and smallest values in a[i : j], respectively.
5.  {
6.     if (i = j) then max := min := a[i]; // Small(P)
7.     else if (i = j – 1) then // Another case of Small(P)
8.        {
9.           if (a[i] < a[j]) then
10.             {
11.             max := a[j]; min := a[i];
12.             }
13.          else
14.          {
15.                max := a[i]; min := a[j];
16.          }
17.       }
18.       else
19.       { // If P is not small, divide P into subproblems.
20.       // Find where to split the set.
21.          mid := ⌊(i + j)/2⌋;
22.       // Solve the subproblems.
23.          MaxMin(i, mid, max, min);
24.          MaxMin(mid + 1, j, max1, min1);
25.       // Combine the solutions.
26.          if (max < max1) then max := max1;
27.          if (min > min1) then min := min1;
28.    }
29.}
```

Q.34. What is the data structures used to perform recursion ? ***(R.G.P.V., Dec. 2016)***

Ans. Stack is used in recursion. Because the stack have the property of last in first out (LIFO) which is required to store the current state of function and jump to new instance of that function. So that when the execution of new instance get over, it jump back to its previous instance.

Q.35. Explain briefly binary search technique.

Or

Explain any one application that can be solved by divide and conque

(R.G.P.V., Dec. 201

Ans. Let a_i, $1 \le i \le n$, be a list of elements that are sorted in nondecreasi order. Now, consider the problem of finding whether a given element x present in the list. If x is present, we are to determine a value j such th $a_j = x$. If x is not in the list, then j is set to zero. Let P = (n, a_i, ..., a_l, x) deno an arbitrary instance of the above problem where n is the number of eleme in the list, a_i, ..., a_l is the list of elements, and x is the element searched fo

Divide and conquer technique is used to solve this problem. Consider sm (P) be true, if n = 1. In this case, S(P) will take the value i if $x = a_i$; otherwise it w take value 0. Then $g(1) = \Theta(1)$. If P has more than one element, then it can further divided into a new subproblem as follows. Pick an index q (in the range l]) and compare x with a_q. There are three possibilities that can occur –

(i) $x = a_q$: In this case the problem P is immediately solved.

(ii) $x < a_q$: In this case x has to be searched for only in the sublist a_{i+1}, ..., a_{q-1}. Therefore, P reduces to (q – i, a_i, ..., a_{q-1}, x).

(iii) $x > a_q$: In this case the sublist to be searched is a_{q+1}, ..., a_l reduces to (l – q, a_{q+1}, ..., a_l, x).

In this example, the problem P gets divided into one new subproble This division takes only $\Theta(1)$ time. After a comparison with a_q, the insta remaining to be solved (if any) can be easily solved by using divide conquer technique again. If q is always chosen such that a_q is the mid element (that is $q = \lfloor (n + 1)/2 \rfloor$), then the resulting search algorithm is cal binary search. Here point to be noted is that the answer to the new subprob is also the answer to the original problem P, there is no need for any combini

Q.36. How divide and conquer technique can be applied to binary tre Also write algorithm for divide and conquer. (R.G.P.V., June 2015, 20

Ans. Refer to Q.35 and Q.32.

Q.37. Give example of binary search technique.

Ans. Let we have the following 14 data items –

– 15, – 6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151 and p them in a [1 : 14], and simulate the steps that BinSearch goes through searches for different values of x. The variables **low, high** and **mid** need t only traced when we execute the algorithm. We try the following values 151, – 14 and 9 for two successful and one unsuccessful search. Table explains the traces of BinSearch on these three inputs.

Table 1.1 Three Examples of Binary Search on 14 Elements

x = 151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found
x = – 14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found
x = 9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			found

Q.38. What is the complexity of binary search algorithm ?

Ans. Binary search technique is beneficial to use when data in an array is stored in increasing numerical order, or equivalently alphabetically.

The complexity of any algorithm is measured by the number f(n) of comparisons to locate ITEM in DATA where DATA contains n elements.

In binary search, each comparison reduces the sample size in half.

Suppose f(n) is the maximum number of comparisons required to locate ITEM in DATA, where

$2^{f(n)} > n$ or

we can say,

$f(n) = [\log_2 n] + 1$.

i.e., the worst time complexity is approximately equal to $\log_2 n$. In the best case complexity can be given as $\theta(1)$.

As in the best case, we find that the data item for which we are searching is the middle term of the data segment.

So, in the first comparison we will result with required location, i.e., MID of data segment. So, the best case complexity is (1).

Also, the average case complexity in successful search can be given as $\Theta[\log n]$. While, in unsuccessful search, complexity in all best, average, and worst case is $\Theta[\log n]$.

Finally, complexity of binary search algorithm in different types of searches (i.e., successful search and unsuccessful search) can be given as –

Successful Searches –

Best Case	Θ(1)
Average Case	Θ[log n]
Worst Case	Θ[log n].

Unsuccessful Searches –

Best Case	Θ[log n]
Average Case	Θ[log n]
Worst Case	Θ[log n]

Q.39. Implement an algorithm for binary search. Discuss in detail abo time complexity of binary search algorithm. (R.G.P.V., May 201

Ans. **Implementation of an Algorithm for Binary Search** – Refer to Prob

Time Complexity of Binary Search Algorithm – Refer to Q.38.

Q.40. Write and explain an algorithm to search an item in array usi divide and conquer strategy with complexity $O(\log_2 n)$. (R.G.P.V., Dec. 201

Ans. Refer to Q.32 and Q.38.

Q.41. Give the divide and conquer solution for binary search and analy its complexity. (R.G.P.V., May 201

Ans. Refer to Q.35 and Q.38.

Q.42. Explain divide and conquer technique. Design a recursi algorithm for binary search. (R.G.P.V., June 2009, 201

Ans. **Divide and Conquer Technique** – Refer to Q.32.

Recursive Algorithm for Binary Search – Algorithm 1.7 describes t recursive binary search procedure, where BinSrch has four inputs a [ ], and x. It is initially invoked as BinSrch (a, *l*, n, x).

Algorithm 1.7 Recursive Binary Search

```
1.  Algorithm BinSrch (a, i, l, x)
2.  //Given an array a [i : l] of elements in nondecreasing.
3.  //Order, 1 ≤ i ≤ l, determine whether x is present, and
4.  // if so, return j such that x = a [j]; else return 0.
5.  {
6.  if (l = i) then // If small (P)
7.  {
8.      If (x = a[i]) then return i;
9.      else return 0;
10. }
11. else
```



```
12. {// Reduce P into a smaller subproblem.
13.     mid : = [(i + l)/2];
14.     if (x = a [mid]) then return mid;
15.     else if (x < a [mid]) then
16.     return BinSrch (a, i, mid – 1, x);
17.     else return BinSrch (a, mid +1, l, x);
18. }
19. }
```

Q.43. Discuss in which condition binary search is better than linear search. (R.G.P.V., June 2016)

Ans. Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be sorted one. The binary search algorithm becomes relatively more efficient when the number of components on the list is large. Binary search is more efficient than linear search because it requires less number of comparisons.

Sequential search (or linear search) is easy to write and efficient for short lists, but a disaster for long ones. Imagine trying to find the name in a large telephone book by reading one name at a time at the front of the book. To find any item in a long sorted list, there are far more efficient methods. One of the best is binary search, in which first to compare the item with one in the center of the list and then restrict our attention to only the first or second half of the list, depending on whether the item comes before or after the central one. In this way, at each step we reduce the length of the list to be searched by half. In only twenty comparisons this method will locate any requested name in a list of a million names.

Q.44. What are the limitations of binary search algorithm ?

Ans. The algorithm requires two conditions –

(i) The list must be sorted and

(ii) One must direct access to the middle element in any sublist.

This implies that one must essentially use a sorted array to hold the data. But keeping data in a sorted array is normally very expensive when there are many insertions and deletions. In such situations, accordingly, one may use a different data structure, like linked list or a binary search tree, to store the data.

Q.45. Write a short note on mergesort. (R.G.P.V., Dec. 2016)

Ans. **MERGING** – Suppose A is a sorted list with r elements and B is a sorted list with s elements.

The operation that combines the elements of A and B into a single sorted list C with n = r + s elements is called ***merging.***

Mergesort – The mergesort closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows –

Divide – Divide the n elements sequence to be sorted into two subsequences of n/2 elements each.

Conquer – Sort the two subsequences recursively using mergesort.

Combine – Merge the two sorted subsequences to produce the sorted answer.

Here, one thing is to be noted that the recursion "bottoms out" when the sequences to be sorted has length 1, in which case there is no work to be done, since every subsequence of length 1 is already in sorted order.

Merging of two sorted subarrays can be done as follows –

Suppose one is given two sorted decks of cards. The decks can be merged as shown in fig 1.18.

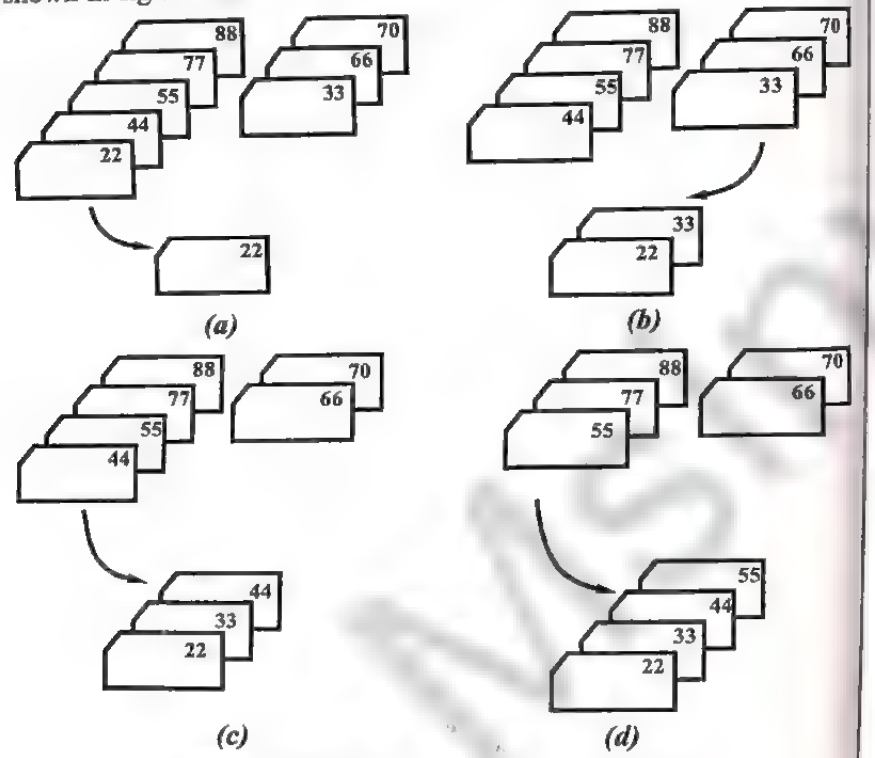


Fig. 1.18 Merging of Two Sorted Arrays

In each step of merging two sorted subarrays, two front cards are compared and the smaller one is placed in the combined deck.

As in fig. 1.18 (a) front cards 22 and 33 are compared, but 22 < 33 so 22 is placed on combined deck.

In each step we do the same i.e., in next step we compare 44 and 33 and 44 > 33, so 33 is placed in the combined deck as shown in fig. 1.18 (b) and so on.

When one of the deck is empty, all the remaining cards in the other deck are put at the end of the combined deck.

And finally we result with a sorted array.

Q.46. Write Mergesort algorithm and analyse your algorithm of different case.

Ans. Mergesort algorithm is given in algorithm 1.8.

Algorithm 1.8 MergeSort Algorithm

```
1. Algorithm MergeSort(low, high)
2. // a[low : high] is a global array to be sorted.
3. // Small(P) is true if there is only one element
4. // to sort. In this case the list is already sorted.
5. {
6.     if (low < high) then // If there are more than one element.
7.     {
8.         // Divide P into subproblems.
9.             // Find where to split the set
10.                mid := ⌊(low + high)/2⌋;
11.        // Solve the subproblems.
12.                MergeSort(low, mid);
13.                MergeSort(mid + 1, high);
14.        // Combine the solutions.
15.                Merge(low, mid, high);
16.    }
17. }
```

Algorithm of merging two sorted subarrays is given in algorithm 1.9.

Algorithm 1.9 Merging Two Sorted Subarrays, Using Auxiliary Storage

```
1. Algorithm Merge(low, mid, high)
2. // a[low : high] is a global array containing two sorted
3. // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4. // is to merge these two sets into a single set residing
5. // in a[low : high]. b[] is an auxiliary global array.
6. {
7.     h := low; i := low; j := mid + 1;
8.     while((h ≤ mid)  and (j ≤ high) do
9.     {
```

```
10.    if(a[h] ≤ a[j]) then
11.  {
12.        b[i] := a[h]; h := h + 1;
13.    }
14.    else
15.    {
16.        b[i] := a[j]; j :=j + 1;
17.    }
18.    i := i + 1;
19.  }
20.  if(h> mid) then
21.    for k := j to high do
22.    {
23.        b[i] := a[k]; i := i + 1;
24.    }
25.  else
26.    for k := h to mid do
27.    {
28.        b[i] := a[k]; i := i + 1;
29.    }
30.  for k := low to high do a[k] := b[k];
31.}
```

Refer to Q.48 for analysis of complexity.

Q.47. Write the mergesort algorithm and discuss its efficiency. Sort E, X, A, M, P, L, E in alphabetical order using mergesort.

(R.G.P.V., May 20

Ans. **Mergesort Algorithm** – Refer to Q.46.

Efficiency of Mergesort Algorithm – The computer takes a time produce the output for a given input, according to the designed algorithm a specific task. The computing efficiency is more important for an algorith it calculate the total time taken by the computer to perform a task. For examp if computer A is 1000 times faster than computer B, means computer processes 1 billion of instructions per second, and computer B processes million of instructions per second. If we run the selection sort algorithm w the order of growth at n^2 and the another sorting algorithm called the 'merges algorithm' with the order of growth at $n \log_2 n$. The mergesort algorithm w be more efficient as compared to any other algorithm. If we run both algorith to sort 1 million of values in an input array, the mergesort algorithm is mo efficient, it takes less time and produce the output.

Let we have to sort data array

E, X, A, M, P, L, E

Pass 1	E, X	A, M	L, P	E
Pass 2	A, E, M, X	E, L, P		
Pass 3	A, E, E, L, M, P, X			

Finally the sorted data is A, E, E, L, M, P, X.

Q.48. Evaluate the time complexity of mergesort.

Ans. In mergesort our recurrence-based analysis is simplified if we assume that the original problem size is a power of two i.e., $n = 2^k$. Where n is total number of elements in array, and k is some constant.

Each divide step yields two subsequences of size exactly n/2.

Mergesort on just one element takes constant time. But when we have $n > 1$ elements, we break down the running time as follows –

Divide – The divide step just computes the middle of the subarray, which takes constant time.

i.e., $\Theta(1)$ when $n = 1$.

Conquer – We recursively solve two subproblems, each of size $\frac{n}{2}$, which contributes $2T\left(\frac{n}{2}\right)$ to the running time.

Combine – Merge procedure on an n elements subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

So, the worst case running time T(n) of merge sort is given as –

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

We solve above complexity by substitution method so as to find actual complexity.

$\Theta(1)$ and $\Theta(n)$ indicate some constant value, with $\Theta(n)$ some function of n, let C.n. So, we can write complexity function T(n) as follows –

Put $\Theta(1) = a$ and $\Theta(n) = C.n$

$$T(n) = \begin{cases} a & n = 1 \\ 2T\left(\frac{n}{2}\right) + C.n & n > 1 \end{cases}$$

Here, a and C are constant and $n = 2^k$.

$$T(n) = 2T\left(\frac{n}{2}\right) + C.n$$

Substituting value of $T\left(\frac{n}{2}\right)$.

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + C.\frac{n}{2}\right] + C.n$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2C.\frac{n}{2} + C.n = 2^2 T\left(\frac{n}{2^2}\right) + 2\,C.n$$

Again substitute value of $T\left(\frac{n}{2^2}\right)$

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + C.\frac{n}{2^2}\right] + C.n$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3\,C.n$$

$\vdots \quad \vdots$

$(k-1)^{th}$ term can be given as –

$$T(n) = 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + (k-1).C.n$$

Similarly, k^{th} term can be given as –

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k.C.n$$

We know that $n = 2^k$

Substituting value of n

$$T(n) = 2^k T\left(\frac{2^k}{2^k}\right) + k.C.n = 2^k T(1) + k.C.n.$$

We know that when

$n = 1,\ T(1) = a.$

So, $T(n) = 2^k a + k.C.n.$

also $n = 2^k$

So, $T(n) = a.n + k.C.n.$

We can also write

$k = \log_2 n \quad [\because n = 2^k]$

So, $T(n) = a.n + n.C.\log_2 n$

or, we can write $T(n) = \Theta(n \log_2 n)$

i.e., complexity as a function of $n \log_2 n$.

So, the complexity of mergesort algoirthm can be given as –

$$\Theta(n \log_2 n).$$

Q.49. Explain mergesort algorithm with example.

Ans. Refer to Q.46.

Let we have to sort data array –

A = (5, 2, 4, 6, 1, 3, 2, 6) by applying mergesort algorithm.

This can be done in the following steps –

(i) Divide the array in two subarrays of size n/2, i.e., 4.

We result with two unsorted subarrays.

(5, 2, 4, 6) and (1, 3, 2, 6).

(ii) Now, again apply mergesort for sorting subarrays separately, resulting in 4 unsorted subarrays

(5, 2), (4, 6), (1, 3), and (2, 6).

(iii) Again apply mergesort on above four subarrays resulting 8 sorted subarrays of length 1 only

(5), (2), (4), (6), (1), (3), (2), and (6).

(iv) Apply merging of subarrays to form new sorted array as shown in fig. 1.19.

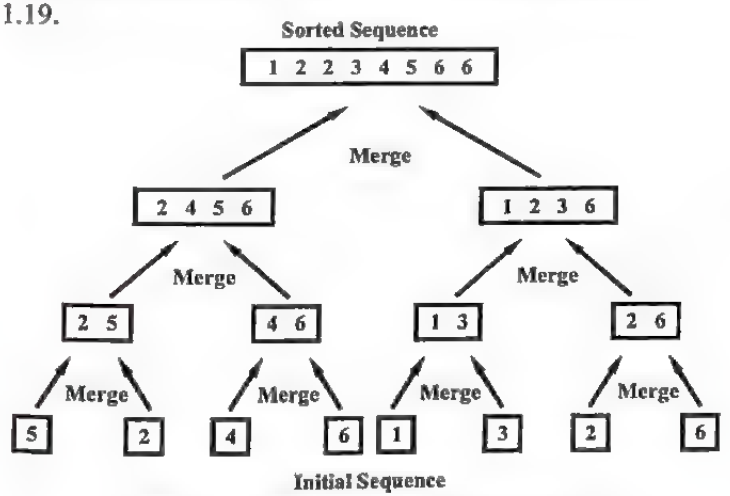


Fig. 1.19 Merging of Sorted Subarrays

(v) Finally we result with a sorted array

(1, 2, 2, 3, 4, 5, 6, 6).

Q.50. Define and explain mergesort algorithm. (R.G.P.V., June 2015)

Ans. Refer to Q.45 and Q.49.

Q.51. Show that the running time of mergesort algorithm on the n-element sequence is O(n log n), even when n is not a power of 2. (R.G.P.V., Dec. 2008)

Ans. If the time for the merging operation is proportional to n, then the computing time for mergesort is described by the recurrence relation is

$$T(n) = \begin{cases} a & n = 1,\ a \text{ a constant} \\ 2T(n/2) + cn & n > 1,\ c \text{ a constant} \end{cases}$$

We start by assuming that this bound holds for (n/2), that is, that $T(n/2) \le c\ (n/2) \log (n/2)$ substituting into the recurrence yields

$$\begin{aligned} T(n) &\le 2(c\ (n/2) \log (n/2)) + cn \\ &\le cn \log (n/2) + cn \\ &= cn \log n - cn \log 2 + cn \\ &= cn \log n - cn + cn = cn \log n \end{aligned}$$

Thus, $T(n) = O(n \log n)$

Q.52. What are the limitations of the basic mergesort algorithm ?

Ans. One complaint, we might raise concerning mergesort is its use of 2n locations. The additional n locations were needed because we could not reasonably merge two sorted sets in place. But despite the use of this space the algorithm must still work hard and copy the result placed into b[low : high] back into a[low : high] on each call of merge. An alternative to this copying is to associate a new field of information with each key. This field is used to link the keys and any associated information together in a sorted list. Then the merging of the sorted lists proceeds by changing the link values, and no records need be moved at all. A field that contains only a link will generally be smaller than an entire record, so less space will be used.

Another complaint we could raise about mergesort is the stack space that is necessitated by the use of recursion. Since mergesort splits each set into two approximately equal-sized subsets, the maximum depth of the stack is proportional to log n. The need for stack space seems indicated by the top-down manner in which this algorithm was devised. The need for stack space can be eliminated if we build an algorithm that works bottom-up.

Q.53. Write a short note on quicksort. (R.G.P.V., May 2018)

Ans. Quicksort, like mergesort, is based on the divide-and-conquer paradigm. Here is the three-steps divide-and-conquer process for sorting a typical subarray A[p,....., r].

Divide – The array A[p,.....,r] is partitioned (rearranged) into two nonempty subarrays A[p,.....,q) and A[q + 1,.....,r] such that each element of A[p,.....,q] is less than or equal to each element of A[q + 1,.....,r]. The index q is computed as part of this partitioning procedure.

Conquer – The two subarrays A[p,.....,q] and A[q + 1,.....,r] are sorted by recursive calls to quicksort.

Combine – Since the subarrays the sorted in place, no work is needed to combine them. The entire array A[p,.....,r] is now sorted.

Q.54. Explain how to apply the divide and conquer strategy for sorting the elements using quicksort ? (R.G.P.V., June 2014, Dec. 2017)

Ans. The divide-and-conquer approach can be used to arrive at an efficient sorting method quicksort. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in a[1 : n] such that a[i] ≤ a [j] for all i between 1 and m and all j between m + 1 and n for some m, 1 ≤ m ≤ n. Thus, the elements in a [1 : m] and a [m + 1 : n] can be independently sorted. No merge is required. The rearrangement of the elements is accomplished by picking some element of a [ ], say t = a [s], and then reordering the other elements so that all elements appearing after t are greater than or equal to t. This rearranging is known as partitioning.

Q.55. Write quicksort algorithm. Discuss its time complexity. (R.G.P.V., Dec. 2011)

Or

Write the complete algorithm for quicksort. Calculate its time complexity in worst, best and average case. (R.G.P.V., June 2008, 2009)

Ans. Refer to Q.53.

Quicksort algorithm is given as follows –

Algorithm 1.10 QuickSort Algorithm

```
1. Algorithm QuickSort(p, q)
2. // Sorts the elements a[p],....., a[q] which reside in the global
3. // array a[1 : n] into ascending order; a[n + 1] is considered to
4. // be defined and must be ≥ all the elements in a[1 : n]
5. {
6.    if (p < q) then // If there are more than one element
7.    {
8.       // Divide P into two subproblems.
9.          j := Partition (a, p, q +1);
```

```
10.                    // j is the position of the partitioning element.
11.              // Solve the subproblems.
12.           QuickSort(p, j – 1);
13.           QuickSort (j + 1, q);
14.              // There is no need for combining solutions.
15.       }
16.   }
```

Partitioning algorithm is given as follows –

Algorithm 1.11 Partitioning the Array a[m : p – 1] about a[m]

```
1. Algorithm Partition(a, m, p)
2. // Within a[m], a[m + 1],....., a[p – 1] the elements are
3. // rearranged in such a manner that if initially t = a[m],
4. // then after completion a[q] = t for some q between m
5. // and p – 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6. // for q < k < p. q is returned. Set a[p] = ∞.
7. {
8.    v := a[m]; i := m; j := p;
9.    repeat
10.   {
11.   repeat
12.         i := i +1;
13.     until (a[i] ≥ v);
14.     repeat
15.         j := j – 1;
16.     until (a[j] ≤ v);
17.     if (i < j) then Interchange (a, i, j);
18.   }until (i ≥ j);
19.   a[m] := a[j]; a[j] := v; return j;
20.}
```

Algorithm 1.12 Swapping of Data

```
1. Algorithm Interchange (a, i, j)
2. // Exchange a[i] with a[j].
3. {
4.    p := a[i];
5.    a[i] := a[j]; a[j] := p;
6. }
```


Time Complexity of QuickSort Algorithm –

The running time of QuickSort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning –

(i) Worst-case Partitioning – The worst-case behaviour for QuickSort occurs when the partitioning routine produces one region with n – 1 elements and one with only 1 element. Let us assume that this unbalanced partitioning arises at every step of the algorithm. Since partitioning costs $\theta(n)$ time and $T(1) = \theta(1)$, the recurrence for the running time is

$$T(n) = T(n-1) + \theta(n).$$

To evaluate this recurrence, we observe that $T(1) = \theta(1)$ and then iterate –

$$T(n) = T(n-1) + \theta(n)$$

$$= \sum_{k=1}^{n} \theta(k)$$

$$= \theta\left(\sum_{k=1}^{n} (k)\right) = \theta(n^2)$$

Fig. 1.20 shows a recursion tree for this worst-case execution of QuickSort.

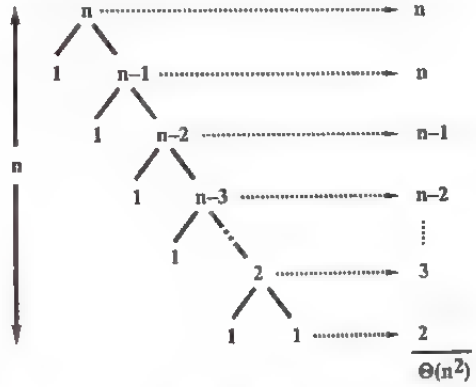


Fig. 1.20

Thus, if the partitioning is maximally unbalanced at every recursive step of the algorithm, the running time is $\theta(n^2)$.

(ii) Best-case Partitioning – If the partitioning procedure produces two regions of size n/2, QuickSort runs much faster. The recurrence is then

$$T(n) = 2T(n/2) + \theta(n)$$

which has solution $T(n) = \theta(n \log n)$.

Fig. 1.21 shows the recursion tree for this best-case execution of QuickSort.

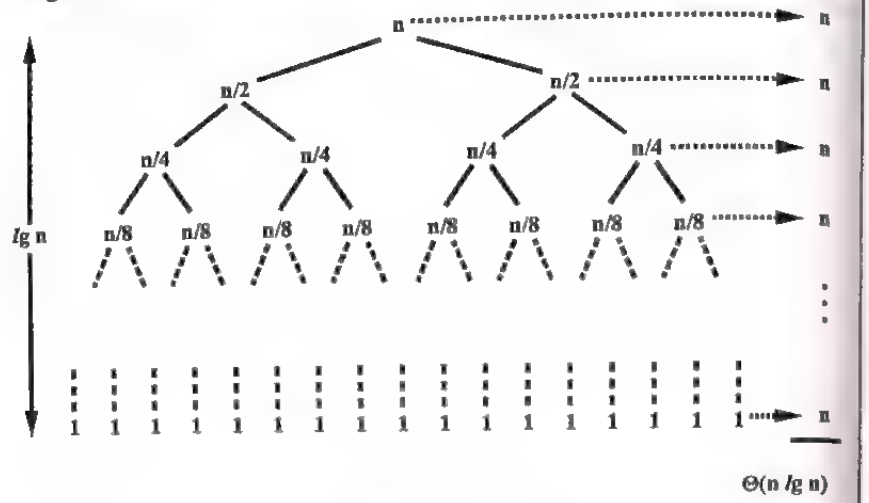


Fig. 1.21

(iii) Balanced Partitioning – Suppose the partitioning algorithm always produces a 9-to-1 proportional split. Then the recurrence relation is

$$T(n) = T(9n/10) + T(n/10) + n$$

of the running time of QuckSort , where θ(n)is replaced by n for convenience. Fig. 1.22 shows the recursion tree for this recurrence. Not that every level of the tree has cost n, until a boundary condition is reached at depth $\log_{10} n = \theta(\log n)$, and then the levels have cost at most n. The recursion terminates at depth $\log_{10/9} n = \theta(\log n)$.

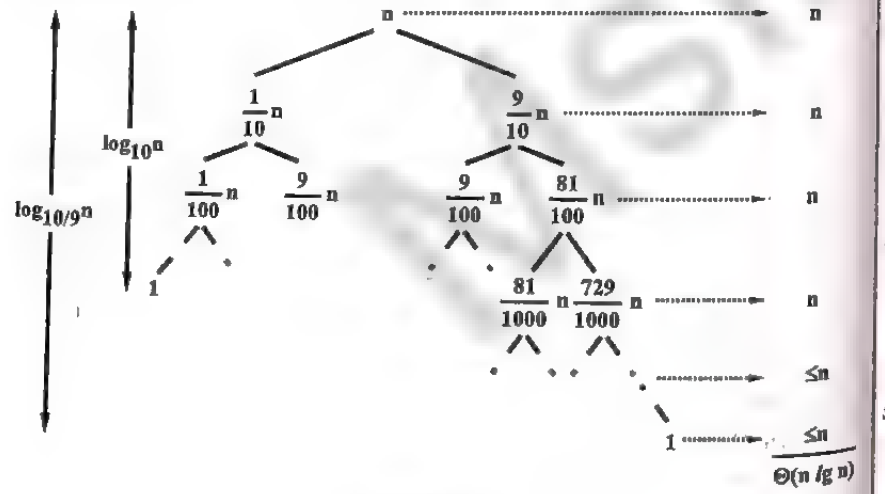


Fig. 1.22

The total cost of QuickSort is therefore θ(n log n). Thus, with a 9-to-1 proportional split at every level of recursion, QuickSort in θ(n log n) time. In fact, even a 99-to-1 split yields an O(n log n) running time. Therefore, the running time is θ(n log n) wherever the split has constant proportionality.

Q.56. What is the running time of quicksort when all elements of array A have the same value ? ***(R.G.P.V., Dec. 2008, June 2010)***

Ans. If all elements of the array A have the same value the partition operation returns $q = \left\lfloor \frac{p-r+1}{2} \right\rfloor$, thus the running time $T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)$. (running time of the Partition operation is Θ(n)). And we usually omit ceilings and floors, hence the recurrence for running time will be $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

We remember such form of recurrences and know that T(n) = O(n lg n) for such a recurrence. Let's prove this bound –

Assume T(n) ≤ c.n.lg n (for n smaller than the given),

then $T(n) \le 2.\left(c.\frac{n}{2}.\lg\frac{n}{2}\right) + \Theta(n) = cn \lg n - cn + \Theta(n) = cn \lg n - cn + cn = cn \lg n$.

Q.57. What are the advantages and disadvantages of quicksort ?

Ans. **Advantages of Quicksort –**

(i) There is good compromise between number of comparisons and data movements.

(ii) This method takes the advantage of initial ordering of the list.

(iii) Average comparisons are less.

Disadvantages of Quicksort –

(i) The behaviour entirely depends upon the selection of pivot (the item where the list is partitioned).

(ii) If pivot is not partitioning the list into approximately equal half, the performance degrades sharply.

(iii) This method uses recursion. Recursion is time consuming.

(iv) The swaps are definitely greater than selection sort swaps.

Q.58. How many average comparisons are required by the following sorting algorithms to sort the only k highest elements, out of n elements ?
Bubble sort, quicksort, insertion sort, mergesort, heapsort.
(R.G.P.V., June 2012)

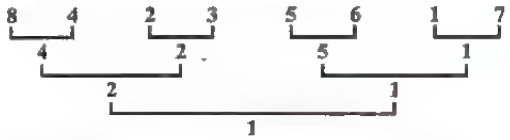
Ans. The table 1.2 presents comparison of various sorting methods.

Table 1.2

Algorithm	*Average*
Bubble sort	$n^2/4$
Insertion sort	$n^2/4$
Quicksort	$O(n \log_2 n)$
Mergesort	$O(n \log_2 n)$
Heapsort	$O(n \log_2 n)$

Q.59. Show that the second smallest of n elements can be found wi n + [log n] – 2 comparisons in the worst case. ***(R.G.P.V., June 201***

Ans. Consider,



Thus, we can have the smallest element in n – 1 comparisons. The le of comparison whose height is $\lceil \lg n \rceil$. We can get the second smallest eleme in $\lceil \lg n \rceil - 1$ comparison. Therefore, total comparisons

$$n + \lceil \lg n \rceil - 2$$

Q.60. What is the purpose of Strassen's matrix multiplication ?
(R.G.P.V., Dec. 201

Ans. The purpose of Strassen's matrix multiplication is to reduce t time complexity of matrix multiplication. It uses the concept that addition two number is faster than its multiplication. So, the time complexity of Strasse matrix multiplication is $O(n^{2.81})$ which is less than the time complexity ordinary matrix multiplication ($O(n^3)$).

Q.61. Write short note on Strassen's matrix multiplication. Compare with conventional divide and conquer technique of matrix multiplicatio
(R.G.P.V., Dec. 201

Or

Explain the Strassen's multiplication technique.
(R.G.P.V., June 2007, 2008, 2010, 2011, 201

Or

Write down Strassen's algorithm for multiplication.
(R.G.P.V., Dec. 2006, 201

Or

Discuss Strassen's algorithm for matrix multiplication with exampl
(R.G.P.V., June 201

Or

Explain Strassen's matrix multiplication with the help of an example.
(R.G.P.V., June 2017)

Or

Explain Strassen's matrix multiplication algorithm.
(R.G.P.V., Dec. 2017)

Or

Write short note on Strassen's matrix multiplication.
(R.G.P.V., Nov. 2018)

Ans. Let A and B be two n × n matrices. The product matrix C = AB is also an n × n matrix whose i, j^{th} element is formed by taking the elements in the i^{th} row of A and j^{th} column of B and then multiplying them to get

$$C(i, j) = \sum_{1 \le k \le n} A(i, k)\, B(k, j) \qquad \ldots(i)$$

for all i and j between 1 and n.

Now to compute C (i, j) using this formula, we need n multiplications. Since the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

The divide-and-conquer strategy suggests another way to compute the product of n × n matrices.

For simplicity assume that n is an exact power of 2, i.e., there exists a non-negative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two. Suppose that A and B are each partitioned into four square submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. Then the product AB can be computed by using the above formula for the product of 2 × 2 matrices – if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \qquad \ldots(ii)$$

where

$$\left.\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}\right\} \qquad \ldots(iii)$$

If n = 2, the formulas (ii) and (iii) are computed using a multiplication operation for the elements of A and B. These elements are typically floating point numbers. For n > 2, the elements of C can be computed using matrix

multiplication and addition operations applied to matrices of size n/2 × n/2. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm using for the n×n case. This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small (e.g. 2) so that the product is computed directly.

To compute AB using equation (iii), we need to perform eight multiplications of n/2 × n/2 matrices and four additions of $\frac{n}{2} \times \frac{n}{2}$ matrices. Since two n/2 × n/2 matrices can be added in time cn^2 for some constant c, the overall computing time T(n) of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved as to obtain $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made.

Since, matrix multiplications are more expensive than matrix additions. So, we reformulate the equations for C_{ij}, with fewer multiplications and possibly more additions.

Volkar Strassen discovered, a way to compute the C_{ij}'s of equation (iii) using only 7 multiplications and 18 additions or subtractions.

For this, first seven $\frac{n}{2} \times \frac{n}{2}$ matrices P, Q, R, S, T, U and V, have to be computed, as given in equation (iv).

$$\left.\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}\right\} \quad \ldots(iv)$$

After this C_{ij} can be computed using equation (v).

$$\left.\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}\right\} \quad \ldots(v)$$

The resulting recurrence relation for T(n) is

$$T(n) = \begin{cases} b & n \le 2 \\ 7T\left(\frac{n}{2}\right) + an^2 & n > 2 \end{cases} \quad \ldots(vi)$$

where, a and b are constants.

Solving above complexity, similar to other recurrence relation, we get

$$T(n) = an^2\left[1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2 + \ldots\ldots\ldots\ldots + \left(\frac{7}{4}\right)^{k-1}\right] + 7^k T(1)$$

$$\le cn^2\left(\frac{7}{4}\right)^{\log_2 n} + 7^{\log 2^n}, c \text{ a constant}$$

$$= cn^{\log 2^4 + \log 2^7 - \log 2^4} + n^{\log 2^7} = O(n^{\log_2 7}) \approx O(n^{2.81}).$$

So, Strassen's matrix multiplication has complexity $O(n^{2.81})$.

Q.62. How can we prove that Strassen's matrix multiplication is advantageous over ordinary matrix multiplication ? (R.G.P.V., May 2019)

Ans. Refer to Q.61.

NUMERICAL PROBLEMS

Prob.6. Solve the recurrence relation –

$$T(n) = 3(n/4) + n \qquad \textit{(R.G.P.V., June 2013)}$$

Or

Solve the following recurrence relation –

$$T(n) = 3T(n/4) + n \qquad \textit{(R.G.P.V., Dec. 2017)}$$

Sol. Given the relation is

$$T(n) = 3(n/4) + n$$

But the recurrence relation should be

$$T(n) = 3T(n/4) + n$$

We iterate is as follows –

$$\begin{aligned} T(n) &= n + 3T(n/4) \\ &= n + 3(n/4 + 3T(n/16)) \\ &= n + 3(n/4 + 3(n/16) + 3T(n/64)) \\ &= n + 3(n/4) + 9(n/16) + 27T(n/64) \end{aligned}$$

where ((n/4)/4) = (n/16) and ((n/16)/4) = (n/64).

The i^{th} term in the series is $3^i(n/4^i)$. The iterations hits $n = 1$ w $(n/4^i) = 1$ or, equivalently, when i exceeds $\log_4 n$. By continuing the itera until this point, we discover that the summation contains a decreasing geome series –

$$T(n) \le n + 3n/4 + 9n/16 + 27n/64 + ... + 3^{\log_4 n}\Theta(1)$$

$$\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) = 4n + o(n) = O(n)$$

Here, we concluded that $3^{\log_4 n} = n^{\log_4 3}$, and we have used the fact $\log_4 3 < 1$ to conclude that $\Theta(n^{\log_4 3}) = o(n)$.

Prob.7. Apply binary search to find 123 in a list –

45, 96, 105, 121, 145, 192, 199, 205, 245, 275, 123, 850, 905.

(R.G.P.V., June 2

Sol. Given data is not in sorted form and the requirement of binary se is – the data must be sorted.

So before performing the binary search, first we have to sort the da

(i) Sort the given data using mergesort.

(a) Divide the array in two subarrays of size n/2 use this until we got one element in every single subarray.

45, 96, 105, 121, 145, 192 , 199, 205, 245, 275, 123, 850, 9

45, 96, 105 , 121, 145, 192 , 199, 205, 245 , 275, 123, 850, 90

45 , 96, 105 , 121 , 145, 192 , 199 , 205, 245 , 275, 123 , 850, 905

45 , 96 , 105 , 121 , 145 , 192 , 199 , 205 , 245 , 275 , 123 , 850 , 905

(b) Now, use backward approach to merge & sort the data

45 , 96, 105 , 121 , 145, 192 , 199 , 205, 245 , 123, 275 , 850, 90

45, 96, 105 , 121, 145, 192 , 199, 205, 245 , 123, 275, 850, 90

45, 96, 105, 121, 145, 192 , 123, 199, 205, 245, 275, 850, 90

45, 96, 105, 121, 123, 145, 192, 199, 205, 245, 275, 850, 905

(ii) Now apply binary search.

Given data is in array which is sorted in increasing numerical order

45, 96, 105, 121, 123, 145, 192, 199, 205, 245, 275, 850, 905

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

and we have to search – 123

So first initialize the variable

BEG = LB = 0

END = UB = 12

MID = INT((BEG + END)/2)

= INT((0 + 12)/2)

= INT(6) = 6

Now we check if DATA[MID] = ITEM

DATA[MID]= DATA[6] = 192

ITEM = 123

and

Clearly DATA[MID]≠ ITEM

Since, DATA[MID]> ITEM

So, updated variable will be

BEG = LB = 0

END = UB = MID – 1 = 6 – 1 = 5

MID = INT((BEG + END)/2)

= INT((0 + 5)/2)

= INT(2.5) = 2

Now,

DATA [MID]= DATA[2] = 96

which is clearly not equal to ITEM

Clearly DATA [MID] < ITEM

So, BEG = LB = MID + 1 = 2 + 1 = 3

END = UB = 5

MID = INT ((BEG + END)/2)

= INT((3 + 5)/2)

= INT(4) = 4

Now, DATA[MID]= DATA[4] = 123

and

ITEM = 123

Clearly DATA[MID]= ITEM

Hence, search successful. ITEM is found at position 4 in the data after sorting.

Prob.8. Write the algorithm for binary search. Apply binary search to find 122 in a list –

44, 95, 104, 120, 144, 191, 198, 204, 244, 274, 122, 849, 904.

(R.G.P.V., Nov. 2018)

Sol. Refer to Q.42.

Given data is not in sorted form and the requirement of binary search is the data must be sorted.

So before performing the binary search, first we have to sort the da

(i) Sort the given data using mergesort.

(a) Divide the array in two subarrays of size n/2 use this until we got one element in every single subarray.

44, 95, 104, 120, 144, 191 , 198, 204, 244, 274, 122, 849, 90

44, 95, 104 , 120, 144, 191 , 198, 204, 244 , 274, 122, 849, 90

44 , 95, 104 , 120 , 144, 191 , 198 , 204, 244 , 274, 122 , 849, 90

44 , 95 , 104 , 120 , 144 , 191 , 198 , 204 , 244 , 274 , 122 , 849 , 90

(b) Now, use backward approach to merge & sort the data

44 , 95, 104 , 120 , 144, 191 , 198 , 204, 244 , 122, 274 , 849, 90

44, 95, 104 , 120, 144, 191 , 198, 204, 244 , 122, 274, 849, 90

44, 95, 104, 120, 144, 191 , 122, 198, 204, 244, 274, 849, 9

44, 95, 104, 120, 122, 144, 191, 198, 204, 244, 274, 849, 904

(ii) Now apply binary search.

Given data is in array which is sorted in increasing numerical order

44, 95, 104, 120, 122, 144, 191, 198, 204, 244, 274, 849, 904

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

and we have to search – 122

So first initialize the variable

BEG = LB = 0

END = UB = 12

MID = INT((BEG + END)/2)

= INT((0 + 12)/2)

= INT(6) = 6

Now we check if DATA[MID] = ITEM

DATA[MID]= DATA[6] = 191

and ITEM = 122

Clearly DATA[MID]≠ ITEM

Since, DATA[MID]> ITEM

So, updated variable will be

BEG = LB = 0

END = UB = MID – 1 = 6 – 1 = 5

MID = INT((BEG + END)/2)

= INT((0 + 5)/2)

= INT(2.5) = 2

Now,

DATA [MID]= DATA[2] = 95

which is clearly not equal to ITEM

Clearly DATA [MID] < ITEM

So, BEG = LB = MID + 1 = 2 + 1 = 3

END = UB = 5

MID = INT ((BEG + END)/2)

= INT((3 + 5)/2)

= INT(4) = 4

Now, DATA[MID]= DATA[4] = 122

and ITEM = 122

Clearly DATA[MID]= ITEM

Hence, search successful. ITEM is found at position 4 in the data after sorting.

Prob.9. Write the procedure of mergesort and sort the given array of 8 elements step-by-step using mergesort 35, 18, 7, 12, 5, 23, 16, 3. ***(R.G.P.V., June 2016)***

Ans. Refer to Q.46.

The list A contains 8 elements as follows –

35, 18, 7, 12, 5, 23, 16, 3

The process of merge sort is illustrated below –

Pass I – Following lists are obtained –

18, 35 , 7, 12 , 5, 23 , 3, 16

Pass II – Following two sorted sublists are –

7, 12, 18, 35 , 3, 5, 16, 23

Pass III – Single sorted list is obtained after merging the two sorted sublists.

3, 5, 7, 12, 16, 18, 23, 35

Prob.10. Discuss the procedure for mergesort. Illustrate how the worst case performance of mergesort is better than the quicksort. Sort the given array of 8 elements step-by-step using mergesort –

A = [36, 19, 8, 13, 6, 24, 17, 4] ***(R.G.P.V., Nov. 2***

Sol. Refer to Q.46, Q.48 and Q.55.

The list A contains 8 elements as follows –

36, 19, 8, 13, 6, 24, 17, 4

The process of merge sort is illustrated below –

Pass I – Following lists are obtained –

$\underbrace{19, 36}, \underbrace{8, 13}, \underbrace{6, 24}, \underbrace{4, 17}$

Pass II – Following two sorted sublists are –

$\underbrace{8, 13, 19, 36}, \underbrace{4, 6, 17, 24}$

Pass III – Single sorted list is obtained after merging the two sorted subl

4, 6, 8, 13, 17, 19, 24, 36

Prob.11. Sort the given list using mergesort –

70, 80, 40, 50, 60, 12, 35, 95, 10 (R.G.P.V., Dec. 2008, June 20

Or

Sort the list 70, 80, 40, 50, 60, 12, 35, 95, 10 by using mergesort.

(R.G.P.V., Dec. 20

Sol. The array of 9 elements a [1 : 9] = (70, 80, 40, 50, 60, 12 35, 95, is given. Algorithm mergesort splits a [ ] into two subarrays a [1 : 5] a[6 : 9]. The elements in a [1 : 5] are then split into two subarrays of size th (a [1 : 3]) and two (a [4 : 5]). Then the items in a[1 : 3] are split into subarr of size two (a[1: 2]) and one (a[3 :3]). The two values in a [1 : 2] are spl final time into one-element subarrays and now the merging begins. Pictoria the file can now be viewed as

(70 | 80 | 40 | 50, 60 | 12, 35, 95, 10)

Elements a[1] and a[2] are merged to yield (70, 80 | 40 | 50, 60| 12, 95, 10). Then a[3] is merged with a[1 : 2] and (40, 70, 80 | 50,60 | 12, 95,10) is produced. Next elements a[4] and a[5] are merged.

(40, 70, 80 | 50, 60 | 12, 35, 95, 10)

and then a [1 : 3] and a [4 : 5]

(40, 50, 60, 70, 80 | 12, 35, 95, 10)

Now, the mergesort processes the second recursive call. Repeat recursive calls produces the following subarrays

(40, 50, 60, 70, 80 | 12, 35 | 95, 10)

Elements a[6] and a]7] are merged. Next a[8] and a[9] are merged, and then a [6 : 7] and a [8: 9]

(40, 50, 60, 70, 80 | 10, 12, 35, 95)

At this point there are two sorted subarrays and the final merge produces the fully sorted result.

(10, 12, 35, 40, 50, 60, 70, 80, 95).

Prob.12. Sort the following list using quicksort technique and argue upon its running time

A = [5, 7, 9, 4, 10, 2, 8, 1] ***(R.G.P.V., Dec. 2015)***

Sol. The given list of elements A is as follows –

5, 7, 9, 4, 10, 2, 8, 1

The reduction step of quicksort algorithm finds the final position of the numbers; procedure steps are as follows –

(5)	7	9	4	10	2	8	(1) ←
(1) →	7	9	4	10	2	8	(5)
1	(5)	9	4	10	2	8	(7) ←
1 →	(2)	9	4	10	(5)	8	7
1	2	(5)	4	10	(9)	8	7 ←
1	2	(4)	(5)	10	9	8	7

Number 5 is placed in its final position. We have two sublists, one left of 5 and second right of 5.

$\underbrace{1\ \ 2\ \ 4}_{\text{First Sublist}}\ \ (5)\ \ \underbrace{10\ \ 9\ \ 8\ \ 7}_{\text{Second Sublist}}$

Now the task of sorting the original list has been reduced to the task of sorting each of the above sublists. The first sublist is already sorted. Applying same quicksort algorithm recursively to second sublist, results in a completely sorted list i.e.

1, 2, 4, 5, 7, 8, 9, 10

Prob.13. Sort the following list using quicksort –

36, 95, 42, 12, 08, 66, 72, 55 ***(R.G.P.V., June 2017)***

Sol. Suppose A is the following list –

(36), 95, 42, 12, 08, 66, 72, (55)

Scan the list from right to left, comparing each number with 36 and s where the element is less than 36. The number is 8. Interchange 36 and obtain the list.

(08), 95, 42, 12, (36), 66, 72, 55

Beginning with 08, scan left to right and stop where the element is gre than 36. The number is 95. Interchange 36 and 95 to obtain the list.

08, (36), 42, 12, (95), 66, 72, 55

Now, applying the same procedure while 36 not placed in correct posi we get two sublists.

08, (12), 42, (36), 95, 66, 72, 55

08, 12, (36), 42, 95, 66, 72, 55

Now do quick sort parallely in each sublist.

08, 12, 36, (42), 95, 66, 72, 55

08, 12, 36, 42, 95, 66, 72, 55

08, 12, 36, 42, (95), 66, 72, (55)

08, 12, 36, 42, 55, 66, 72, (95)

08, 12, 36, 42, (55), 66, 72, 95

08, 12, 36, 42, 55, (66), 72, 95

08, 12, 36, 42, 55, 66, 72, 95

Sorted List – 8, 12, 36, 42, 55, 66, 72, 95

Prob.14. Apply quicksort to sort the following list –

36, 12, 85, 79, 46, 18, 92, 30, 28, 65, 72 ***(R.G.P.V., May 20***

Sol. Suppose A is the following list –

(36), 12, 85, 79, 46, 18, 92, 30, 28, 65, (72)

Scan the list from right to left, comparing each number with 36 and s where the element is less than 36. The number is 28. Interchange 36 and 2 obtain the list.

(28), 12, 85, 79, 46, 18, 92, 30, (36), 65, 72

Beginning with 28, scan left to right and stop where element is greater than 36. The number is 85. Interchange 36 and 85 to obtain the list.

28, 12, (36), 79, 46, 18, 92, 30, (85), 65, 72

Now beginning with 85, scan the list right to left, comparing each number with 36 and stop where the element is less than 36. The number is 30. Interchange 36 and 30 to be obtain the list.

28, 12, (30), 79, 46, 18, 92, (36), 85, 65, 72

Now beginning with 30, scan left to right and stop where the element is greater than 36. The number is 79. Interchange 36 and 79 to obtain the list.

28, 12, 30, (36), 46, 18, 92, 79, 85, 65, (72)

Now, applying the same procedure while 36 not placed in correct position, we get two sublists.

28, 12, 30, (36), 46, 18, 92, 79, 85, 65, (72)

← less than 36

28, 12, 30, (18), 46, (36), 92, 79, 85, 65, 72

28, 12, 30, 18, (36), (46), 92, 79, 85, 65, 72

→ greater than 36

Now 36 is at its final position, and the list is divided in two sublists.

28, 12, 30, 18, (36), 46, 92, 79, 85, 65, 72

I II

Now, we sort the above first sublist repeating the same procedure.

(28), 12, 30, (18)

← less than 28

[Interchange 28 and 18]

(18), 12, 30, (28)

→ greater than 28 [Interchange 28 and 30]

18, 12, (28), (30)

(18), (12), 28, 30 [Interchange 18 and 12]

Now 28 is at its proper place and sorted sublist is

12, 18, 28, 30

Now we sort the second sublist repeating the same procedure.

(92), 79, 85, 65, (72)

→ less than 92

(72), 79, 85, (65), 92

→ greater than 65

65, (79), 85, (72), 92

65, 72, (85), (79), 92

65, 72, 79, 85, 92

And sorted sublist is

65, 72, 79, 85, 92

Hence, the original, sorted list will be

12, 18, 28, 30, 36, 46, 65, 72, 79, 85, 92

Prob.15. Apply quicksort algorithm for the following array and sort element (Take first element of the list as the pivot element).

24, 56, 47, 35, 10, 90, 82, 31.

Also discuss complexity of algorithm. ***(R.G.P.V., Dec. 20***

Sol. Pivot = 24.

Scan the list from right to left, comparing each number with pivot, 24, and stop where the element is less than 24. The number is 10. Interchan 24 and 10. Now the list will be

(10), 56, 47, 35, (24), 90, 82, 31

Beginning with 10, scan left to right and stop where the element is gre than 24. The number is 56.

Interchange 56 and 24 to obtain the list.

10, (24), 47, 35, 56, 90, 82, 31

Now, 24 is placed at its correct position.

So, the list is divided into two sublists.

10, (24), 47, 35, 56, 90, 82, 31

Sublist Sublist

Apply, same procedure on two sublists concurrently.

Pivot

10, 24, (47), 35, 56, 90, 82, (31)

10, 24, (31), 35, 56, 90, 82, (47)

Sorted

10, 24, 31, 35, (56), 90, 82, (47)

10, 24, 31, 35, (47), 90, 82, 56

10, 24, 31, 35, 47, (90), 82, (56)

10, 24, 31, 35, 47, 56, 82, 90 **Ans.**

Complexity of Quicksort – Refer to Q.55.

Prob.16. If $A = \begin{bmatrix} 5 & 3 & 0 & 2 \\ 4 & 3 & 2 & 6 \\ 7 & 8 & 1 & 4 \\ 9 & 4 & 6 & 7 \end{bmatrix}$, $B = \begin{bmatrix} 3 & 2 & 4 & 7 \\ 2 & 5 & 2 & 9 \\ 3 & 9 & 0 & 3 \\ 7 & 6 & 2 & 1 \end{bmatrix}$

Implement Strassen's matrix multiplication on A and B.

(R.G.P.V. Dec. 2008)

Sol. Given two matrices of 4 × 4 and hence its product will be 4 × 4. Therefore $n = 4 = 2^2$.

Now these two matrices are partitioned into four square submatrices, each submatrix having dimensions 2 × 2.

Thus, the product A × B = C can be calculated as follows –

$$A = \begin{bmatrix} \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 0 & 2 \\ 2 & 6 \end{bmatrix} \\ \begin{bmatrix} 7 & 8 \\ 9 & 4 \end{bmatrix} & \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix} \end{bmatrix}, \quad B = \begin{bmatrix} \begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} & \begin{bmatrix} 4 & 7 \\ 2 & 9 \end{bmatrix} \\ \begin{bmatrix} 3 & 9 \\ 7 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix} \end{bmatrix}$$

Now

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$$= \left(\begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}\right)\left(\begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} + \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 6 & 7 \\ 10 & 10 \end{bmatrix}\begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} = \begin{bmatrix} 46 & 72 \\ 70 & 110 \end{bmatrix}$$

$Q = (A_{21} + A_{22})B_{11}$

$$= \left(\begin{bmatrix} 7 & 8 \\ 9 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}\right)\begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 12 \\ 15 & 11 \end{bmatrix}\begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 48 & 76 \\ 67 & 85 \end{bmatrix}$$

$R = A_{11}(B_{12} - B_{22})$

$$= \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix}\left(\begin{bmatrix} 4 & 7 \\ 2 & 9 \end{bmatrix} - \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}\right) = \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix}\begin{bmatrix} 4 & 4 \\ 0 & 8 \end{bmatrix} = \begin{bmatrix} 20 & 44 \\ 16 & 40 \end{bmatrix}$$

$$S = A_{22}(B_{21} - B_{11})$$

$$= \begin{bmatrix}1 & 4\\6 & 7\end{bmatrix}\cdot\left[\begin{bmatrix}3 & 9\\7 & 6\end{bmatrix}-\begin{bmatrix}3 & 2\\2 & 5\end{bmatrix}\right] = \begin{bmatrix}1 & 4\\6 & 7\end{bmatrix}\cdot\begin{bmatrix}0 & 7\\5 & 1\end{bmatrix} = \begin{bmatrix}20 & 11\\35 & 49\end{bmatrix}$$

$$T = (A_{11} + A_{12})\,B_{22}$$

$$= \left[\begin{bmatrix}5 & 3\\4 & 3\end{bmatrix}+\begin{bmatrix}0 & 2\\2 & 6\end{bmatrix}\right]\cdot\begin{bmatrix}0 & 3\\2 & 1\end{bmatrix} = \begin{bmatrix}5 & 5\\6 & 9\end{bmatrix}\cdot\begin{bmatrix}0 & 3\\2 & 1\end{bmatrix} = \begin{bmatrix}10 & 20\\18 & 27\end{bmatrix}$$

$$U = (A_{21} - A_{11})\,(B_{11} + B_{12})$$

$$= \left[\begin{bmatrix}7 & 8\\9 & 4\end{bmatrix}-\begin{bmatrix}5 & 3\\4 & 3\end{bmatrix}\right]\cdot\left[\begin{bmatrix}3 & 2\\2 & 5\end{bmatrix}+\begin{bmatrix}4 & 7\\2 & 9\end{bmatrix}\right]$$

$$= \begin{bmatrix}2 & 5\\5 & 1\end{bmatrix}\cdot\begin{bmatrix}7 & 9\\4 & 14\end{bmatrix} = \begin{bmatrix}34 & 88\\39 & 59\end{bmatrix}$$

$$V = (A_{12} - A_{22})\,(B_{21} + B_{22})$$

$$= \left[\begin{bmatrix}0 & 2\\2 & 6\end{bmatrix}-\begin{bmatrix}1 & 4\\6 & 7\end{bmatrix}\right]\cdot\left[\begin{bmatrix}3 & 9\\7 & 6\end{bmatrix}+\begin{bmatrix}0 & 3\\2 & 1\end{bmatrix}\right]$$

$$= \begin{bmatrix}-1 & -2\\-4 & -1\end{bmatrix}\cdot\begin{bmatrix}3 & 12\\9 & 7\end{bmatrix} = \begin{bmatrix}-21 & -26\\-21 & -55\end{bmatrix}$$

$$C_{11} = P + S - T + V$$

$$= \begin{bmatrix}46 & 72\\70 & 110\end{bmatrix}+\begin{bmatrix}20 & 11\\35 & 49\end{bmatrix}-\begin{bmatrix}10 & 20\\18 & 27\end{bmatrix}+\begin{bmatrix}-21 & -26\\-21 & -55\end{bmatrix} = \begin{bmatrix}35 & 37\\66 & 77\end{bmatrix}$$

$$C_{12} = R + T = \begin{bmatrix}20 & 44\\16 & 40\end{bmatrix}+\begin{bmatrix}10 & 20\\18 & 27\end{bmatrix} = \begin{bmatrix}30 & 64\\34 & 67\end{bmatrix}$$

$$C_{21} = Q + S = \begin{bmatrix}48 & 76\\67 & 85\end{bmatrix}+\begin{bmatrix}20 & 11\\35 & 49\end{bmatrix} = \begin{bmatrix}68 & 87\\102 & 134\end{bmatrix}$$

$$C_{22} = P + R - Q + U$$

$$= \begin{bmatrix}46 & 72\\70 & 110\end{bmatrix}+\begin{bmatrix}20 & 44\\16 & 40\end{bmatrix}-\begin{bmatrix}48 & 76\\67 & 85\end{bmatrix}+\begin{bmatrix}34 & 88\\39 & 59\end{bmatrix} = \begin{bmatrix}52 & 128\\58 & 124\end{bmatrix}$$

Hence, the product C is

$$= \begin{bmatrix}\begin{bmatrix}35 & 37\\66 & 77\end{bmatrix} & \begin{bmatrix}30 & 64\\34 & 67\end{bmatrix}\\ \begin{bmatrix}68 & 87\\102 & 134\end{bmatrix} & \begin{bmatrix}52 & 128\\58 & 124\end{bmatrix}\end{bmatrix}$$

or $$C = \begin{bmatrix}35 & 37 & 30 & 64\\66 & 77 & 34 & 67\\68 & 87 & 52 & 128\\102 & 134 & 58 & 124\end{bmatrix}$$

UNIT 2

STUDY OF GREEDY STRATEGY, EXAMPLES OF GREEDY METHOD LIKE OPTIMAL MERGE PATTERNS, HUFFMAN CODING, MINIMUM SPANNING TREES, KNAPSACK PROBLEM, JOB SEQUENCING WITH DEADLINES, SINGLE SOURCE SHORTEST PATH ALGORITHM

Q.1. Explain about greedy technique. ***(R.G.P.V., June 2015)***

Or

Explain the concept behind greedy strategy. ***(R.G.P.V., Dec. 2015)***

Or

What is greedy approach ? ***(R.G.P.V., June 2016)***

Ans. The greedy method says that one can devise an algorithm that works in stages considering one input at a time. A decision is made regarding whether a particular input is in an optimal solution at each step. This done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself depends on some optimization measure. This measure may be the objective function. In fact, several different optimization measures may be possible for a given problem. However, most of these will result in algorithms that generate suboptimal solutions. This version of greedy technique is known as the subset paradigm.

Q.2. What is an optimal solution ?

Or

What do you mean by feasible solution ? ***(R.G.P.V., Dec. 2014)***

Ans. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to get a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution.

Q.3. What is greedy techniques ? Derive the equation for the optimal solutions. ***(R.G.P.V., Dec.***

Ans. Refer to Q.1.

Equation for Optimal Solution – If A is the set of all feasible solutions of any problem f( ). Then the optimal solution will be –

max(f(a)) or min(f(a))

where a ∈ A

Q.4. Write the general characteristics of greedy algorithm. ***(R.G.P.V., June 2014, Dec.***

Ans. Some of the general properties of greedy methods are –

(i) Greedy-choice Property – The first key ingredient is the greedy-choice property. A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. Here is where greedy algorithms differ from dynamic programming. In dynamic programming we make a choice at each step, but the choice may depend on the solutions to subproblems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

(ii) Optimal Substructure – A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

Q.5. Explain the optimal merge pattern briefly. ***(R.G.P.V., June 2008,***

Ans. Optimal merge pattern is a pattern that relates the merging of two or more sorted files in a single sorted file.

If we have two sorted files containing n and m records respectively they could be merged together, to obtain one sorted file in time O (n + m).

For example, if files x_1, x_2, x_3 and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then y_1 and x_3 are merged to get y_2. Finally y_2 and x_4 are merged to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1 and then x_3 and x_4 to get y_2. Finally y_1 and y_2 are merged to get the desired sorted file.

There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require different amounts of computing time.

The problem we address ourselves is that of finding an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. As this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Q.6. Write and explain the greedy strategy for optimal merge patterns.

Or

How two way merge pattern can be represented by binary merge tree ? ***(R.G.P.V., June 2016)***

Or

Explain the concept behind greedy strategy. Using optimal merge pattern greedy method, merge the following files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively. ***(R.G.P.V., Nov. 2018)***

Ans. According to greedy method, since merging m-records file and n-records file requires possible m + n record moves, the obvious choice for a selection criterion is – at each step merge two smallest size files together. Thus if we have five files (x_1, x_2, ..., x_5) with sizes (20, 30, 10, 5, 30), our Greedy rule would generate the following merge patterns –

(i) Merge two files, i.e., x_3 and x_4 with record length 10 and 5 respectively, resulting in a file z_1 ($|z_1| = 15$).

(ii) Merge z_1 with x_1 ($|x_1| = 20$) to result a file z_2 ($|z_2| = 35$).

(iii) Now, merge files x_2 and x_5 to get z_3 ($|z_3| = 60$).

(iv) Finally merge z_2 and z_3 resulting file z_4 with 95 record movements.

The above method results in total 205 record movements. Above merging can be shown in fig. 2.1.

In the fig. 2.1, we have five external nodes indicating five files, represented by rectangle. All other files are indicated by internal nodes in the form of circles. Each internal node has exactly two children.The external node x_4 is at a distance of 3 from the root z_4 (a node at level i is at a distance of i – 1 from the root). Hence, the records of file x_4 are moved three times, once to get z_1, once again to get z_2, and finally one more time to get z_4.

If d_i is the distance from the root to the external node for file x_i, and q_i the length of x_i, then the total number of record moves for this binary merge tree is –

$$\sum_{i=1}^{n} d_i \, q_i$$

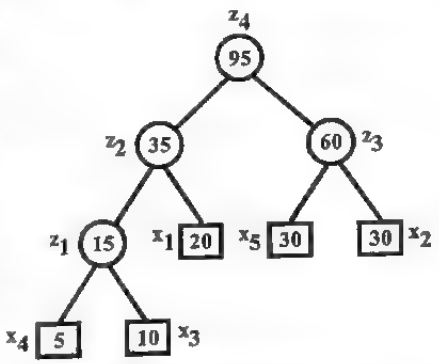


Fig. 2.1 Binary Merge Tree Representing a Merge Pattern

This sum is called the weighted external path length of the tree.

An optimal two-way merge pattern corresponds to a binary merge with minimum weighted external path length. The function tree of algo 2.1 uses the greedy rule to get a two-way merge tree for n files. The algo contains as input alist *list* of n trees. There are three fields, *lchild*, *rchild* weight in each node in a tree. Initially, each tree in list contains just one This external node has *lchild* and *rchild* fields zero whereas weight length of one of the n files to be merged. For any tree in *list* with root no t → weight gives the length of the merged file it represents. There are functions least (list) and insert (list, t) in function tree. least (list) obtains in *list* whose root has least weight and returns a pointer to this tree. This is deleted from list. Function insert (list, t) inserts the tree with root t into

The main for loop in this algorithm is executed n – 1 times. If list is in increasing order according to the weight value in the roots, then least needs only O (1) time and insert (list, t) can be performed in O (n) Hence, the total time taken is O (n^2). If the *list* is represented as a minhe which the root value is less than or equal to the values of its children, least (list) and insert (list, t) can be done in O (log n) time. In this situatio computing time for tree is O (n log n).

Algorithm 2.1 Algorithm to Generate a Two-way Merge Tree

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};
1.  Algorithm Tree(n)
2.  // list is a global list of n single node
3.  // binary trees as described above.
4.  {
5.      for i := 1 to n – 1 do
6.      {
7.          pt := new treenode; // Get a new tree node.
8.          (pt → lchild) := Least(list); // Merge two trees with
9.          (pt → rchild) := Least(list); // smallest lengths.
10.         (pt → weight) := ((pt → lchild) → weight)
11.             + ((pt → rc ild) → weight);
12.         insert(list, pt);
13.     }
14.     return least(list); // Tree left in list is the merge tree.
15. }
```


Q.7. Discuss in brief Huffman codes.

Ans. Binary trees with minimal weighted external path length can be used to obtain an optimal set of codes for messages M_1, ..., M_{n+1}. Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree. A decode tree is a binary tree in which external node represents messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of fig. 2.2 corresponds to codes 000, 001, 01 and 1 for messages M_1, M_2, M_3 and M_4 respectively. These codes are known as Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\Sigma_{1 \le i \le N+1} q_i d_i$, where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length. $\Sigma_{1 \le i \le N+1} q_i d_i$, is also the expected length of a transmitted message. Hence the code that minimizes expected decode time also minimizes the expected length of a message.

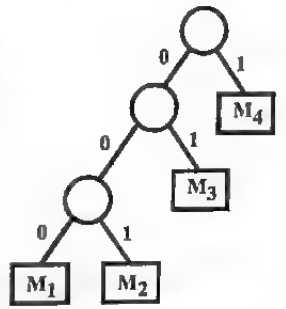


Fig. 2.2 Huffman Codes

Q.8. Define the external path length. ***(R.G.P.V., June 2015)***

Ans. The external path length is the sum of the lengths of all the paths from the root to any external node. The length of the path is the number of its edges.

Q.9. Explain greedy algorithm for constructing a Huffman code. ***(R.G.P.V., June 2010)***

Ans. An optimal prefix code referred to as a ***Huffman code*** is constructed by a greedy algorithm invented by Huffman. The tree T corresponding to the optimal code is built by the algorithm in a bottom-up fashion. The algorithm starts with a set of |C| leaves. It performs a sequence of |C| – 1 merging operations to obtain the final tree.

In the pseudocode, it is assumed that C is a set of n characters in which every character C∈C is an object with a defined frequency f[C]. The two least-frequency objects to merge together are identified using a priority queue PQ, keyed on f. This results in a new object whose frequency is the sum of the frequencies of the two objects merged.

```
HUFFMAN (C)
(i)   n ← |C|
(ii)  PQ ← |C|
(iii) for j ← 1 to n – 1
(iv)  do z ← ALLOCATE-NODE( )
(v)   x ← left[z] ← EXTRACT-MIN(PQ)
(vi)  y ← right[z] ← EXTRACT-MIN(PQ)
(vii) f[z] ← f[x] + f[y]
(viii) INSERT(PQ, Z)
(ix)  return EXTRACT-MIN(PQ)
```

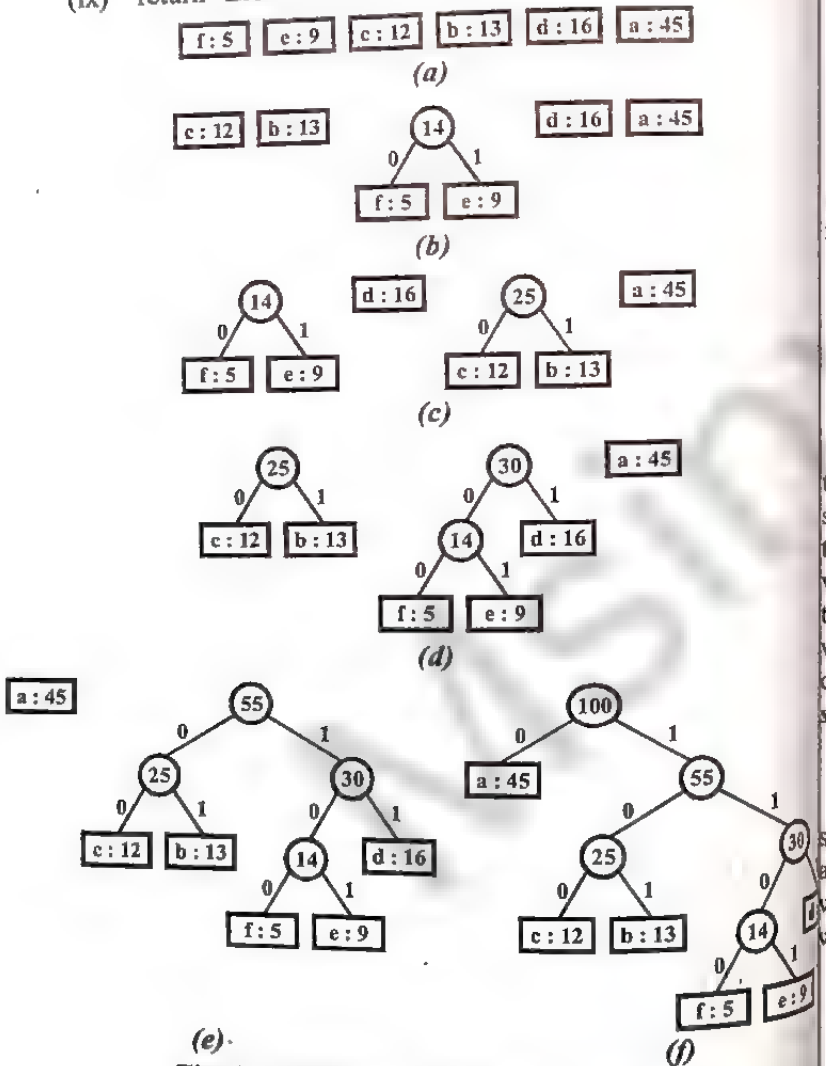


Fig. 2.3 The Steps in Huffman's Algorithm

For example, the working of Huffman's algorithm is as shown in fig. 2.3. The initial queue size n is 6 because there are 6 letters in the alphabet. The five merge steps are needed to build the tree. The optimal prefix code is shown in the final tree. For a letter, the sequence of edge labels on the path from the root to the letter is the codeword.

The initialization of the priority queue PQ with the characters in C is done by give (ii). In lines (iii) – (viii), the for loop repeatedly accesses the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z. Line 7 calculates the frequency of z as the sum of frequencies of x and y. The left and right children of the node z are x and y, respectively. Line (ix) returns the one node left in the queue that is, the root of the code tree, after n – 1 mergers.

It is assumed that PQ is implemented as a binary heap for the analysis of the running time of Huffman's algorithm. The initialization of PQ in line (ii) is done in O(n) time for a set C of n characters. In line (iii) – (viii), the for loop is executed exactly |n| – 1 times. The loop contributes O(n lg n) to the running time, because each heap operation needs time O(lg n). Therefore, the total running time of HUFFMAN (C) on a set of n characters is O(n lg n).

Q.10. What is minimum spanning tree ? ***(R.G.P.V., Dec. 2014)***

Or

Define minimum spanning tree. ***(R.G.P.V., June 2015)***

Or

Write short note on minimum spanning tree. ***(R.G.P.V., Dec. 2017)***

Ans. A spanning tree of a graph G is a minimal subgraph connecting all the vertices of G. If graph "G" is weighted graph then the weight or cost of a spanning tree "T" obtained from a graph "G" is defined as the total sum of all the weights associated with the branches in the given spanning tree "T". As we have observed that there exists several spanning trees of a graph "G" so in the case of weighted graph, different spanning trees of "G" will have different weights. A spanning tree with the minimum weight in a weighted graph is called ***minimal spanning tree*** or ***shortest spanning tree*** or ***minimum cost spanning tree.***

Q.11. What do you mean by minimum spanning tree problem ? Explain.

Ans. A spanning tree for a connected, undirected graph, G = (V, E), is a subgraph of G that is an undirected tree and contains all the vertices of G. In a weighted graph G = (V, E, W), the weight of a subgraph is the sum of the weights of the edges in the subgraph. A minimum spanning tree (MST) for a weighted graph is a spanning tree with minimum weight.

There are two algorithms for solving minimum – spanning tree problem.

(i) Prim's algorithm and (ii) Kruskal algorithm.

Both the algorithms are Greedy approach to the problem.

At each step we try to minimize the cost of the tree or maximize the p
Minimum spanning tree algorithm grows a spanning tree by addi
edge at a time. Fig. 2.4 shows an example of connected graph and its mini
spanning tree.

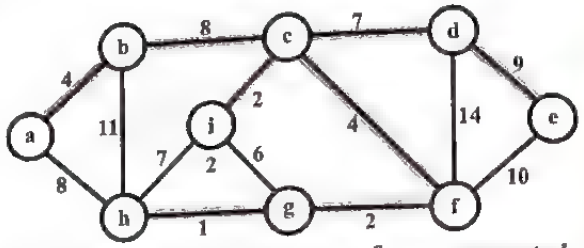


Fig. 2.4 A minimum spanning tree for a connected graph. T
weights on edges are shown, and the edges in a minimum spanning
are shaded. The total weight of the tree shown is 37. The tree is n
unique: removing the edge (b, c) and replacing it with the edge (a, h) yi
another spanning tree with weight 37

Q.12. Write Prim's algorithm to get minimum cost spanning tre
give its complexity.

Or

Explain Prim's algorithm. ***(R.G.P.V., June***

Or

Give an algorithm for computing minimum spanning tree.
(R.G.P.V., June

Ans. Prim's algorithm is a special case of generic minimum-span
tree algorithm.

Prim's algorithm is given in algorithm 2.2.

Algorithm 2.2 Prim's Minimum-cost Spanning Tree Algorithm

```
1.  Algorithm Prim(E, cost, n, t)
2.  // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3.  // adjacency matrix of an n vertex graph such that cost [i, j] is
4.  // either a positive real number or ∞ if no  edge (i, j) exists.
5.  // A minimum spanning tree is computed and stored as a set of
6.  // edges in the array t[1 : n – 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge
7.  // the minimum-cost spanning tree. The final cost is returned.
8.  {
9.      Let (k, l) be an edge of minimum cost in E;
10.     mincost := cost [k, l];
11.     t[1,1] := k; t[1,2] := l;
12.     for i := 1 to n do // Initialize near.
```



```
13.         if(cost[i, l] < cost[i,k]) then near[i] := l;
14.         else near[i] := k;
15.     near[k] := near [l] := 0;
16.     for i := 2 to n – 1 do
17.     { // Find n – 2 additional edges for t.
18.         Let j be an index such that near[j] ≠ 0 and
19.         cost[j, near[j]] is minimum;
20.         t[i,1] := j; t[i,2] := near[j];
21.         mincost := mincost + cost[j, near[j]];
22.         near[j] := 0;
23.         for k := 1 to n do // Update near[].
24.             if ((near [k] ≠ 0) and  (cost[k, near[k]] > cost[k, j]))
25.                 then near[k] := j;
26.     }
27.     return mincost;
28. }
```

In above algorithm we have E i.e., the set of edges in G.

cost [1: n, 1 : n] is the cost adjacency matrix of an n vertex graph such that cost [i, j] is either a positive real number or if no edge (i, j) exists.

A minimum spanning tree is computed and stored as a set of edges in the array t[1 : n – 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. At each step, a light edge connecting a vertex in A to a vertex in V – A is added to the tree. The next edge chosen according to minimize the cost of the tree i.e., an edge that results in a minimum increase of the sum of costs of the edges so far included. The Prim's algorithm will start with a tree that includes only a minimum-cost edge of G.

Then the edges are added to the tree one by one.

The next edge (i, j) to be added in such that i is a vertex already included in the tree. j is a vertex not yet included, and cost of (i, j), cost[i, j] is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree.

For this we make a check for every possible edge that connects a node still not in tree, with a node present in tree for their minimum cost. For determining this new edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value near[j]. The value near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j] we define

near[j] = 0 for all vertices j that are already in the tree. The next edge to inc is defined by the vertex j such that near[j] ≠ 0 (j not already in the tree), cost [j, near[j]] is minimum.

Complexity – The time required by Prim's algorithm is $O(n^2)$, whe is the number of vertices in the graph G. Notice that line 9 takes O(1E1) t and line 10 takes Θ(1) time. The **for** loop of line 12 takes Θ(n) time. Here 18, 19 and the **for** loop of line 23 need O(n) time. So, each iteration of the loop of line 16 takes O(n) time. The total time for the **for** loop of line 16 is $O(n^2)$. Thus, Prim runs in $O(n^2)$ time.

Q.13. Write the Kruskal's algorithm for obtaining minimum spann tree. Calculate its time in worst case. ***(R.G.P.V., Dec. 20***

Or

Define Kruskal's algorithm. Also write down the steps for Krusk algorithm in detail. ***(R.G.P.V., June 20***

Or

Given E is the set of edges in graph G. G has n vertices. Cost [u, v] is cost of edge (u, v). T is the set of edges in the minimum-cost spanning t Write the pseudocode for Kruskal algorithm by considering parame mentioned above. ***(R.G.P.V., June 20***

Ans. Let A be a subset of E that is included in some minimum spann tree for G. In Kruskal's algorithm, the set A is a forest. The safe edge added A is always a least-weight edge in the graph that connects two disti components. While in Prim's algorithm, as we studied earlier, the set A fo a single tree. The safe edge added to A is always a least weight edge connecti the tree to a vertex not in the tree. Kruskal's algorithm is given in algorithm 2

Algorithm 2.3 Kruskal Algorithm

```
1.  Algorithm Kruskal (E, cost, n, t)
2.  // E is the set of edges in G. G has n vertices. cost[u, v] is the
3.  // cost of edge (u, v). t is the set of edges in the minimum-cost
4.  // spanning tree. The final cost is returned.
5.  {
6.      Construct a heap out of the edge costs using Heapify;
7.      for i := 1 to n do parent[i] := – 1;
8.      // Each vertex is in a different set.
9.      i := 0; mincost := 0.0;
10.     while ((i < n – 1) and (heap not empty)) do
11.     {
12.         Delete a minimum cost edge (u, v) from the heap
13.         and reheapify using Adjust;
```



```
14.         j := Find(u); k := Find(v);
15.         if (j ≠ k) then
16.         {
17.             i := i + 1;
18.             t[i,1] := u; t[i, 2] := v;
19.             mincost := mincost + cost[u,v];
20.             Union(j, k);
21.         }
22.     }
23.     if (i ≠ n – 1) then write ("No spanning tree");
24.     else return mincost;
25. }
```

In Kruskal algorithm, at each step, we have to find the minimum cost edge in given graph so as to minimize the cost of the tree.

In Kruskal method, if initially E is the set of edges in graph G.

Then, the two main functions that we wish to perform are –

(i) Determine an edge with minimum cost.

(ii) Delete this edge.

Both of above functions can be performed efficiently if the edges in the E are maintained as a sorted sequential list. For this, edges are maintained as minheap, then the next edge to be considered can be obtained in O(log |E|) time. While the construction of heap itself takes (|E|) time.

Secondly, after determining the next minimum cost edge, we have to make sure that this edge is not going to result in a cycle. If it is resulting a cycle then it will be discarded and new edge will be searched, otherwise it will be added.

For determining this we prepare two sets. One set has all connected components of t. Then, two vertices u and v are connected iff they are in the same set. And the other set have all the components that are not connected to tree t.

On the basis of above two sets, all the edges that connect vertices present in same set are discarded, as they result into a cycle. And the edge that connects the vertices in the different set is included in the tree. As it does not result into a cycle.

The computing time is, therefore, determined by the time for lines 4 and 5, which in the worst case is O (|E| log |E|).

Based on above discussion in algorithm 2.3, we first construct a initial heap of edges in set E as given in line 6.

In line 7, each vertex is assigned to a distinct set (and hence a distinct tree). The set t is the set of edges to be included in the minimum-cost spanning tree and i is the number of edges in t. The set t can be represented as a sequential list using a two-dimensional array t[1: n – 1, 1: 2]. Edge (u, v) can be added to t by the assignments t[i, 1] = u; and t[i, 2] = v.

In the while loop of line 10, edges are removed from the heap one by on nondecreasing order of cost. Line 14 determines the sets containing u and j ≠ k, then vertices u and v are in different sets (and so in different trees) edge (u, v) is included into t. The sets containing u and v are combined (line If u = v, the edge (u, v) is discarded as its inclusion into t would create a cy

Line 23 determines whether a spanning tree was found. It follows i ≠ n – 1 iff the graph G is not connected. The computing time is O(|E| log where E is the edge set of G.

Q.14. What is spanning tree ? Write Kruskals algorithm with an exam to find minimal spanning tree. ***(R.G.P.V., June 20***

Ans. Refer to Q.10, Q.13 and Prob.12.

Q.15. Tabulate the differences between Kruskal's and Prim's algorit ***(R.G.P.V., Dec. 20***

Or

Write the basic difference between Prim's algorithm and Krusk algorithm. ***(R.G.P.V., Dec. 20***

Ans. The difference between Kruskal's and Prim's algorithm are as follow

S.No.	*Kruskal's Algorithm*	*Prim's Algorithm*
(i)	It uses global approach i.e., at every step, it chooses the cheapest available edge anywhere which does not violate the goals of creating a spanning tree.	It uses local approach i.e., at ev step, it chooses the cheapest available edge attached to any previously chosen vertex whic does not violate the goals of creating a spanning tree.
(ii)	It begins with an edge.	It starts with a node.
(iii)	It selects the next edge in hapha-zard way but in increasing order.	It moves from one node to anoth
(iv)	At any point of time, the set of selected edges need not belong to the same tree. It forms forest.	At any point of time, the set of selected edges will form a sing tree.
(v)	Time complexity O(log V) where V = number of vertices	Time complexity = $O(V^2)$
(vi)	Works on both connected or disconnected graph.	It is restricted to work on connect graphs.

Q.16. What is Knapsack problem ? How can we solve using gre approach? ***(R.G.P.V., May 20***

Ans. In Knapsack problem we are given objects, also weight of ea object is given, i.e., object i has weight w_i.

We suppose here that the Knapsack or bag has capacity m. Also we consider that if a fraction x_i, $0 \le x_i \le 1$ of object i is placed into the Knapsack then the profit $p_i x_i$ is earned.

The objective is to fill the Knapsack so as to maximize the total profit earned. With a constraint that the total weight of all the object in Knapsack should be at most m.

The problem can be stated as – Maximize $\sum_{1 \le i \le n} p_i x_i$ ....(i)

Subject to $\sum_{1 \le i \le n} w_i x_i \le m$ ....(ii)

and $0 \le x_i \le 1$, and $1 \le i \le n$ ....(iii)

Here, the profits and weights are positive numbers.

Any set $(x_1,.....,x_n)$ is said to be feasible if it satisfies equation (ii), and said to be optimal if for which equation (i) is maximized. Algorithm of greedy Knapsack is given in algorithm 2.4.

Algorithm 2.4 Algorithm for Greedy Strategies for the Knapsack Problem

```
1.  Algorithm GreedyKnapsack(m,n)
2.  // p[1 : n] and w [1 : n] contain the profits and weights respectively
3.  // of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1]
4.  // m is the Knapsack size and x[1 : n] is the solution vector.
5.  {
6.      for i := 1 to n do x[i] := 0.0; // Initialize x.
7.      U := m;
8.      for i := 1 to n do
9.      {
10.         if (w[i] > U) then break;
11.         x[i] := 1.0; U := U – w[i];
12.     }
13.     if (i ≤ n) then x[i] := U/w[i];
14. }
```

Q.17. What is Knapsack problem in greedy strategy ? Write the running time and recurrence relation of Knapsack algorithm. (R.G.P.V., May 2018)

Ans. **Knapsack Problem** – Refer to Q.16.

The running time of Knapsack algorithm can be considered as the H^{ϵ}. There are mainly the $\binom{n}{l}$ (trivially bounded by $O(n^l)$) executions of all possible

subsets of cardinality l to perform. Since the running time linearly yields (n^{l+1}) time bound. The greedy strategy requires an additional O (n log n) for sorting the items.

It will be desirable to have $\in$ approximation schemes where the run time raises only moderately both with the accuracy and with the numbe items. Thus, the $\in$ approximation schemes can be classified on the influe of $\in$ on their running time. The recurrence relation of Knapsack algorithm be given as –

Consider n objects, where object i has weight wi and value vi (b positive), and a Knapsack with a weight capacity W.

Now, select some of the objects to fill the Knapsack so as to maxi the total value, without violating the weight constraint.

There are two possible greedy heuristics fails for the 0/1 Knaps problem –

(i) Maximize value without regard for weight –

Pack most valuable objects first.

(ii) Maximize value per unit weight –

Pack objects in decreasing order of value/weight ratio.

Q.18. Explain the greedy strategy. Write algorithm for Knapsack probl ***(R.G.P.V., Dec. 2007, 2009, 20***

Or

Explain the greedy strategy. Write algorithm for greedy strategies Knapsack problem. ***(R.G.P.V., June 20***

Ans. **Greedy Strategy** – Refer to Q.1.

Knapsack Problem – Refer to Q.16.

Q.19. Prove that the fractional Knapsack problem has the greedy-cho property. ***(R.G.P.V., June 20***

Ans. Say that the greedy strategy uses the i most expensive items fi and the rest is part of the i + 1st most expensive item. Then suppose so non-greedy strategy worked better. It would then have to use less than all one of the i most expensive items, or less of the i + 1st most expensive ite in order to have space to use some other item. Say that it uses less of ite and that amount more of item k. You could replace that amount of item k w the missing amount of item j and have a mixture that is at least this valua You can repeat this process until all the amounts of items not included in greedy mix but included in the "optimal" non-greedy solution have been repla by amounts of items in the greedy solution; and your greedy mix is at leas valuable. This is a contradiction.

Q.20. Discuss job sequencing problem by an example. ***(R.G.P.V., June 2016)***

Ans. There is a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\Sigma_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value.

Refer to Prob.19.

Q.21. Explain how job sequencing with deadline can be solved using greedy approach. ***(R.G.P.V., June 2014)***

Ans. To formulate a greedy algorithm to obtain an optimal solution, we must formulate an optimization measure to determine how the next job is chosen. As a first attempt, we choose the objective function $\Sigma_{i \in J} p_i$ as our optimization measure. Using this measure, the next job to include is the one that increases $\Sigma_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires the jobs to be in non increasing order of p_i's.

One way to determine whether a given J is a feasible solution, is to try out all possible permutations of the jobs in J and check whether the jobs in J can be processed in any one of these permutations (sequences) without violating the deadlines. For a given permutation $\sigma = i_1, i_2, i_3, \ldots, i_k$, this is easy to do, since the earliest time job i_q, $1 \leq q \leq k$, will be completed is q. If $q > d_{iq}$, then using σ, at least job i_q will not be completed by its deadline. However, if $|J| = i$, this requires checking i! permutations. Actually, the feasibility of a set J can be determined by checking only one permutation of the jobs in J. This permutation is any one of the permutations in which jobs are ordered in non decreasing order of deadlines.

Q.22. Write an algorithm for job sequencing problem with deadline. Also discuss its complexity. ***(R.G.P.V., Dec. 2013)***

Ans. The algorithm is given in algorithm 2.5.

Algorithm 2.5 Greedy Algorithm for Sequencing Unit Time Jobs with Deadlines and Profits.

```
1. Algorithm JS (d, j, n)
2. //d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3. // are ordered such that p[1] ≥ p[2] ≥ ..... ≥ p[n]. J[i]
4. // is the i^th job in the optimal solution, 1 ≤ i ≤ k. Also
5. // at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
```

```
6.  {
7.      d[0]: = J[0] : = 0; //Initialize
8.      J[1]: = 1; // Include Job 1
9.      K: = 1;
10.     for i : = 2 to n do
11.     {
12.         //consider jobs in nonincreasing order of p[i]. Find
13.         //position for i and check feasibility of insertion.
14.         r: = k;
15.         while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r : = r – 1;
16.         if((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17.         {
18.             //Insert i into J[ ]
19.             for q: = k to (r + 1) step – 1 do J[q + 1]: = J[q];
20.             J[r + 1]: = i; k: = k + 1;
21.         }
22.     }
23.     return k;
24. }
```

For JS there are two possible parameters in terms of which its comple can be measured. We can use n, the number of jobs, and s, the number of j included in the solution J. The **while** loop in this algorithm is iterated at m k times. Each iteration takes $\Theta(1)$ time. If the **if** condition is true, then lines and 20 are executed. These lines require $\Theta(k - r)$ time to insert job i. Hen the total time for each iteration of the **for** loop is $\Theta(k)$. This loop is iterat $n - 1$ times. If s is the final value of k i.e., s is the number of jobs in the fi solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \le n$, t worst-case time, as a function of n alone is $\Theta(n^2)$. If the job set is $p_i = d$ $n - i + 1$, $1 \le i \le n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hen the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space need for d, JS needs $\Theta(s)$ amount of space for J.

The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ using the disjoint set union and find algorithms and a different method determine, the feasibility of a partial solution. If J is a feasible subset of jo then we can determine the processing times for each of the jobs using the cr – if job i has not been assigned a processing time, then assign it to the sl $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \le r \le d_i$ and the sl $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much possible.

Q.23. What do you mean by single-source shortest paths problem ? Explain.

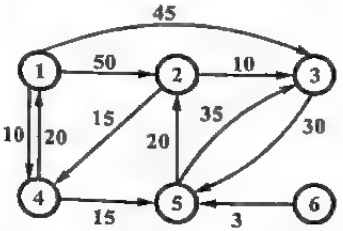
Ans. Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges are assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions –

(i) Is there a path from A to B ?

(ii) If there is one or more paths from A to B, which is the shortest path ?

These problems are the special cases of the path problem. The length of a path is now defined to be the sum of weights of the edges on that path. The starting vertex of the path is referred to as the ***source,*** and the last vertex the ***destination.*** In the problem we are given a directed graph G = (V, E), a weighting function ***cost*** for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to all the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence, this problem fits the ordering paradigm.

For example, consider the directed graph of fig. 2.5 (a). The numbers on the edges are the weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is 1, 4, 5, 2. The length of this path is 10 + 15 + 20 = 45. However, there are three edges on this path, it is shorter than the path 1, 2 which is of length 50. There is no path from 1 to 6. Fig. 2.5 (b) lists the shortest paths from node 1 to nodes 4, 5, 2 and 3 respectively.



(a) Graph

S.No.	Path	Length
1	1, 4	10
2	1, 4, 5	25
3	1, 4, 5, 2	45
4	1, 3	45

(b) Shortest Paths from 1

Fig. 2.5 Graph and Shortest Paths from Vertex 1 to All Destinations

Q.24. Write a greedy-based algorithm to generate the shortest paths.

Or

Write Dijkstra's greedy algorithm to find the shortest path between two given vertices.

Or

Write and explain single source shortest path algorithm with example.

(R.G.P.V., June 2013)

Or

Write algorithm for single source shortest path and find its complexity. ***(R.G.P.V., May 2010)***

Ans. Dijkstra's algorithm only determines the lengths of the shortest paths from v_0 to all other vertices in G. In the function shortest paths it is assumed that the n vertices of G are numbered 1 to n. The set S is maintained as a bit array with S[i] = 0 if vertex i is not in S and S[i] = 1 if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with cost[i, j]'s being the weight of the edge ⟨i, j⟩. The weight cost [i, j] is set to ∞, in case the edge ⟨i, j⟩ is not in E(G). For i = j, cost [i, j] can be set to any nonnegative number without affecting the outcome of the algorithm.

Algorithm 2.6 Greedy Algorithm for Shortest Path

```
1.  Algorithm shortest paths (v, cost, dist, n)
2.  //dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3.  //path from vertex v to vertex j in a diagraph G
4.  //with n vertices.dist[v] is set to zero. G is
5.  //represented by its cost adjacency matrix cost[1 : n, 1 : n].
6.  {
7.  for i: – 1 to n do
8.  { //Initialize S.
9.      S[i]: = false; dist[i]: = cost [v, i];
10. }
11. S[v]: = true; dist[v] = 0.0; //Put v in S.
12. for num: = 2 to n – 1 do
13. {
14.     //Determine n – 1 paths from v.
15.     Choose u from among those vertices not in S such
16.     that dist[u] is minimum;
17.     S[u]: = true; //Put u in S.
18.     for (each w adjacent to u with S[w] = false) do
19.     //update distances
20.         if(dist[w] < (dist[u] + cost[u, w])) then
21.             dist[w]: = dist[u] + cost[u, w];
22.     }
23. }
```

The complexity of the improved version of Dijkstra's algorithm may be realized with a worst-case running time of $\Theta(n_E.\log(n_V))$, an average-case running time of $\Theta(n_V.\log(n_V))$, and a best-case time of $\Theta(n_E)$, with n_E and n_V denoting the number of edges and vertices in the graph, respectively.

Q.25. Write Dijkstra's algorithm. Find the shortest path between S and T in the following graph.

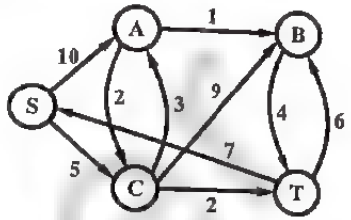


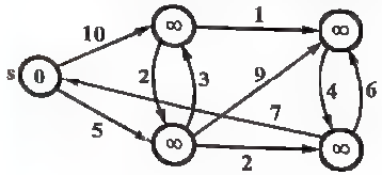
Fig. 2.6 ***(R.G.P.V., Dec. 2011)***

Ans. **Dijkstra's Algorithm** – Dijkstra's algorithm has a set S of vertices whose final shortestpath weights from the source s have already been determined. It means that for all vertices v ∈ S, we have d[v] = δ(s, v). The algorithm repeatedly chooses the vertex u → V – S with the minimum shortest path estimate, inserts u into S, and releases all edges leaving u. In the following implementation, a priority Q that contains all the vertices in V – S, keyed by their d values. It is assumed in the implementation that graph G is represented by adjacency lists.

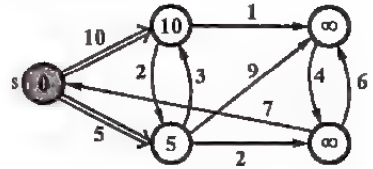
DIJKSTRA's Algorithm –

Initialize-SINGLE-Source (G, s)
S ← 0
Q ← V[G]
while Q ≠ 0
do u ← Extract source MIN (Q)
    S ← S ⋃ {u}
for each vertex v ∈ Adj[u]
    do RELAX (u, v, w)

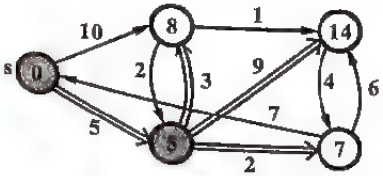
The execution of Dijkstra's algorithm is shown below –



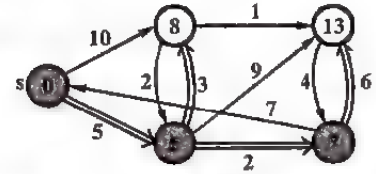
(a)



(b)



(c)



(d)

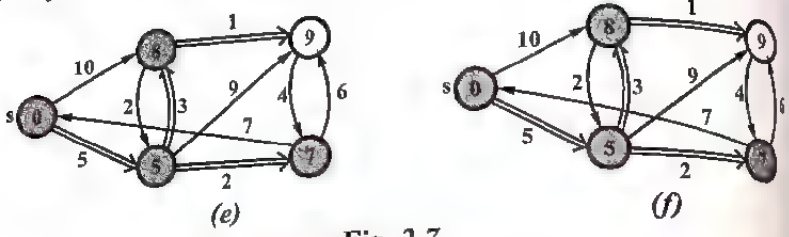


Fig. 2.7

NUMERICAL PROBLEMS

Prob.1. Find the optimal binary merge tree (pattern) for ten files whose length are 28, 32, 12, 5, 84, 53, 91, 35, 3 and 11. Also find its weighted external path length. ***(R.G.P.V., May/June 2006, Dec. 20...***

Sol. For obtaining optimal binary merge pattern, merge two smallest size files at each step.

Fig. 2.8 shows a binary merge pattern representing the optimal merge pattern obtained for the above ten files.

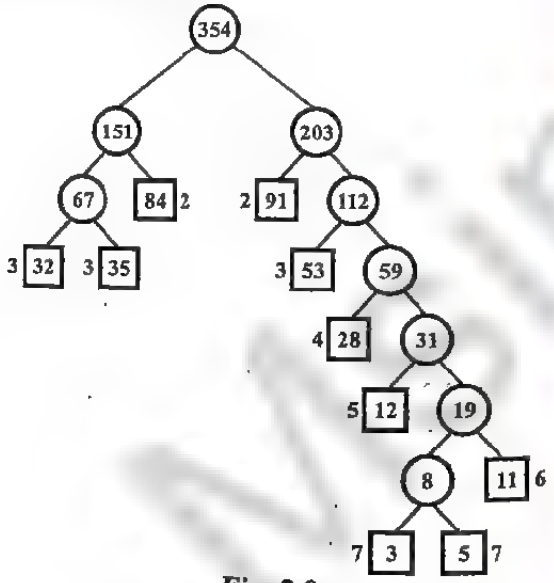


Fig. 2.8

Weighted External Path Length –

$$\sum_{i=1}^{n} d_i q_i$$

where, d_i = Distance from the root to the external node

q_i = The length of x_i.

Here, n = 10

Therefore, $\sum_{i=1}^{10} d_i q_i = d_1q_1 + d_2q_2 + ... + d_{10}\, q_{10}$

$= (4 \times 28) + (3 \times 32) + (5 \times 12) + (7 \times 5) + (2 \times 84) + (3 \times 53)$
$+ (2 \times 91) + (3 \times 35) + (7 \times 3) + (6 \times 11)$

$= 112 + 96 + 60 + 35 + 168 + 159 + 182 + 105 + 21 + 66$

$=$ **1004** **Ans.**

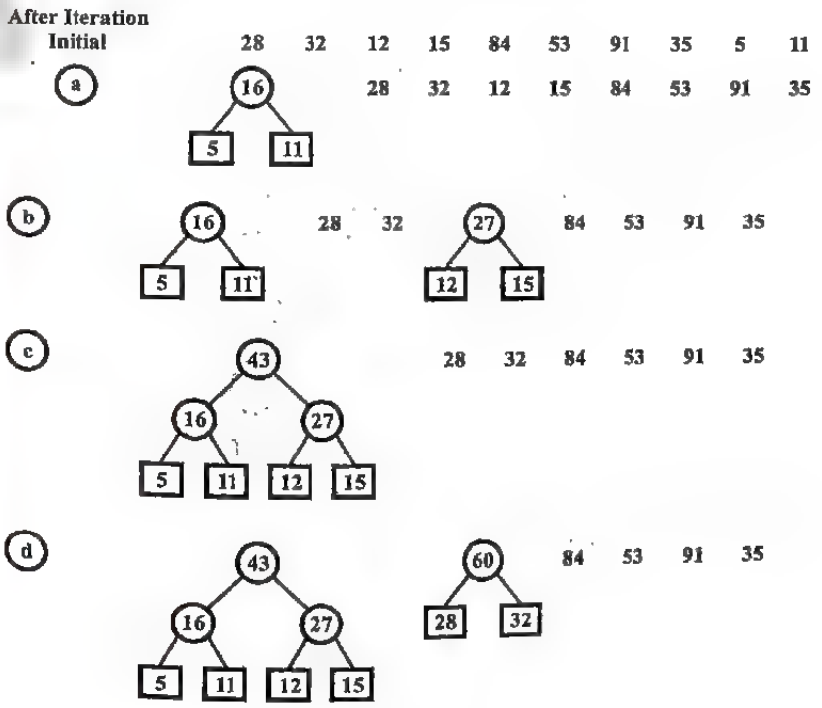
Prob.2. Find the optimal merge pattern for the following data –
28, 32, 12, 5, 84, 53, 91, 35, 3, 11.
(R.G.P.V., Dec. 2008, June 2009, Dec. 2009, 2012, 2017)

Sol. Refer to Prob.1.

Prob.3. What is greedy strategy ? Explain optimal merge pattern in brief and find an optimal binary merge tree (pattern) for files whose lengths are 28, 32, 12, 15, 84, 53, 91, 35, 5 and 11. ***(R.G.P.V., Dec. 2013)***

Sol. Refer to Q.1 and Q.6.

Problem – The optimal merge pattern is given below –



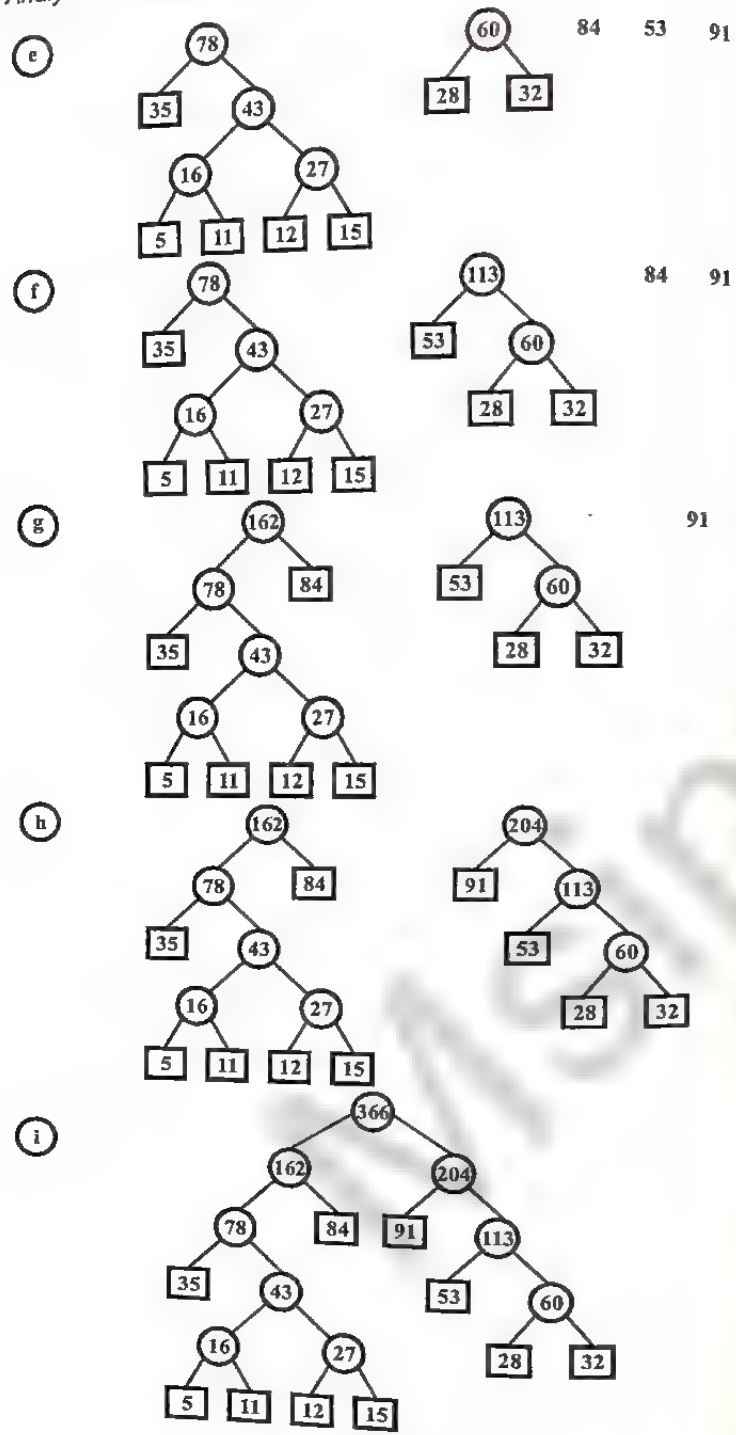
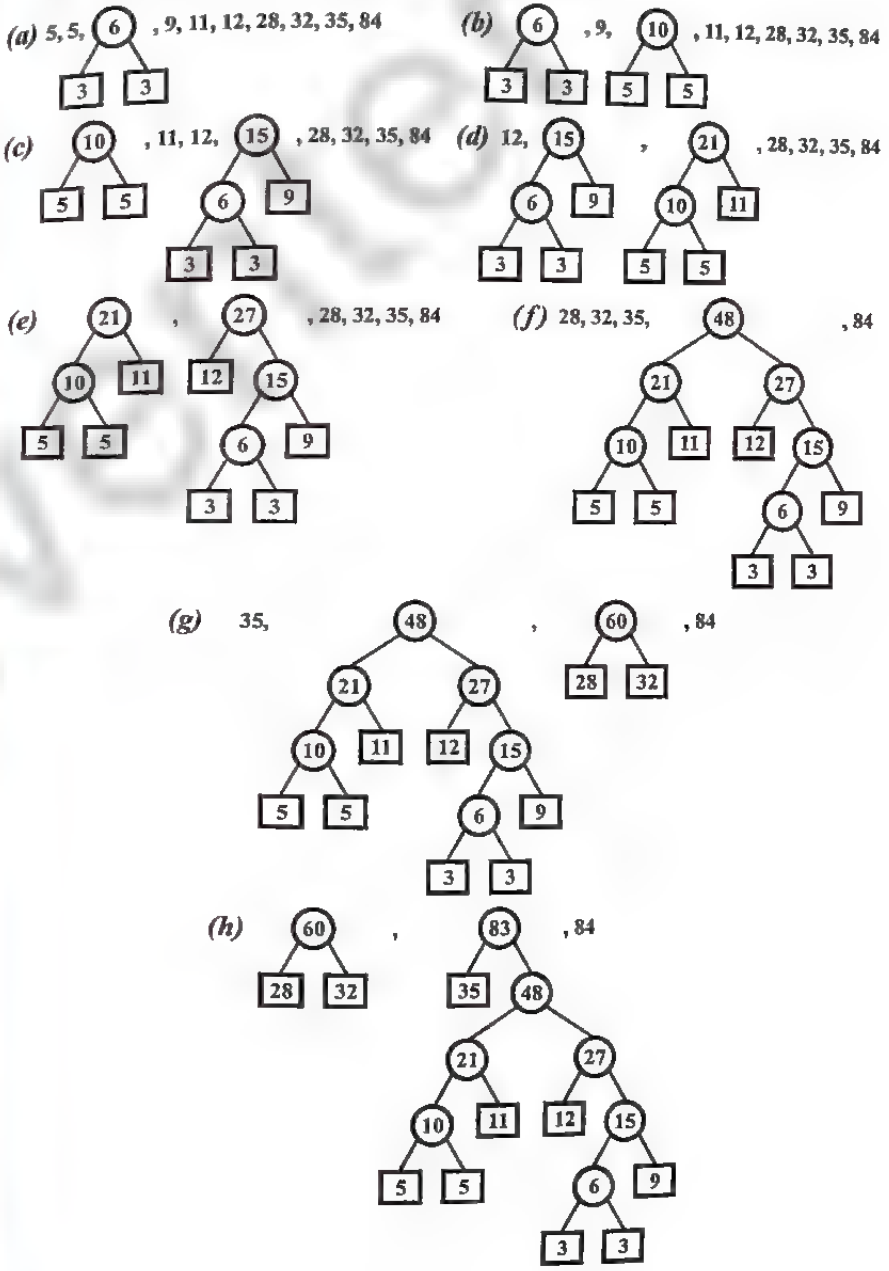


Fig. 2.9

Prob.4. Find an optimal merge pattern for 11 files whose length are 28, 32, 12, 5, 84, 5, 3, 9, 35, 3, 11. Write and explain the algorithm used and determine its complexity. ***(R.G.P.V., Dec. 2015)***

Sol. The optimal merge pattern is given below –



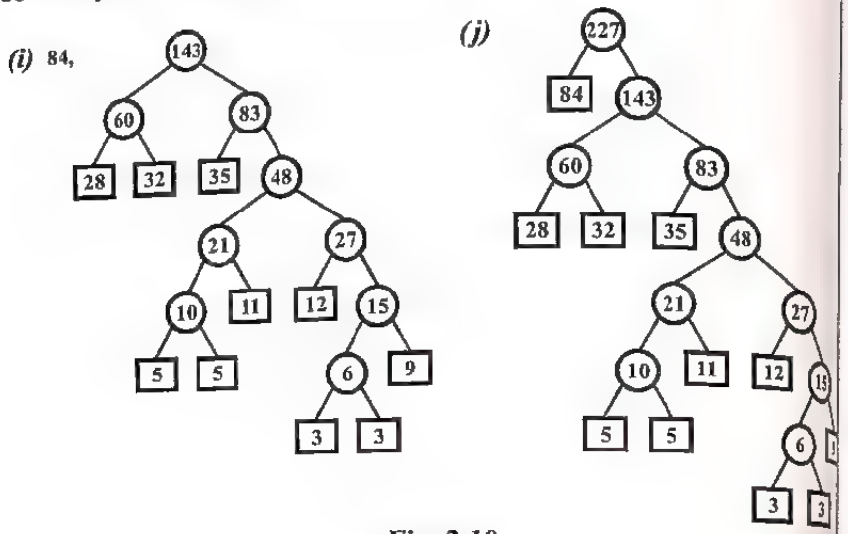


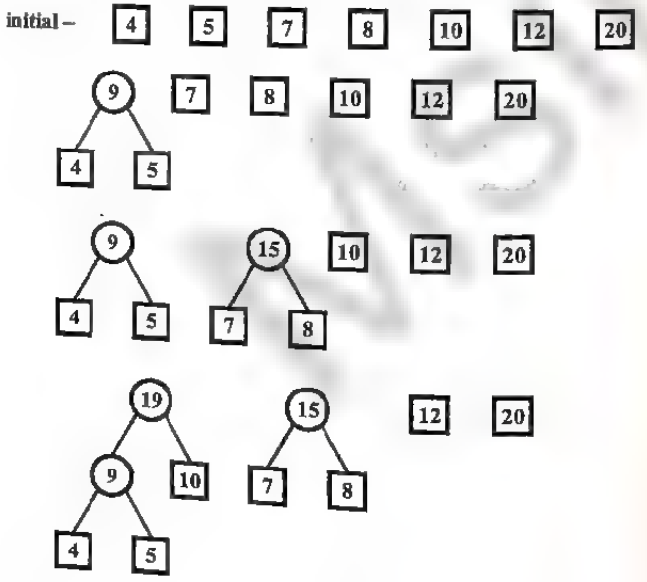
Fig. 2.10

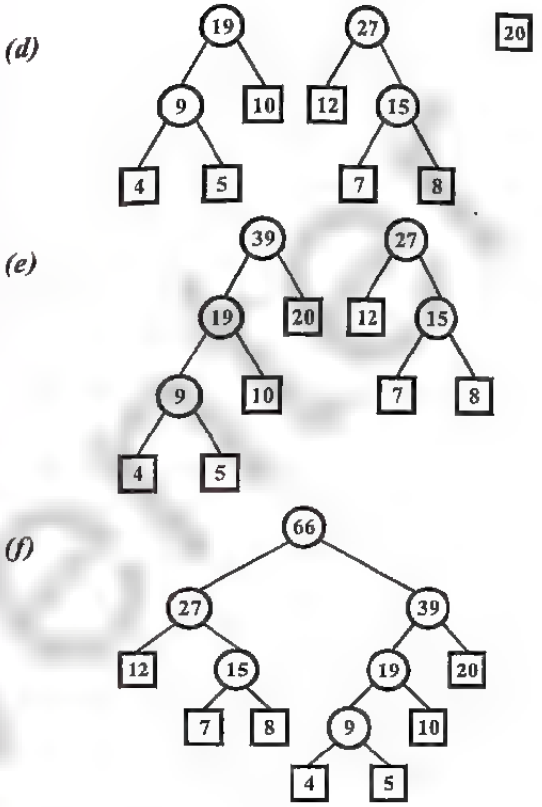
Algorithm and Complexity – Refer to Q.6.

Prob.5. Obtain a set of optimal Huffman codes for the seven messa ***(M_1, ..., M_7) with relative frequencies (q_1,....., q_7) = (4, 5, 7, 8, 10, 12,*** ***Draw the decode tree for this set of codes.***

(R.G.P.V., Dec. 2006, June 2009, Dec. 20

Sol. There are seven messages (M_1.....M_7) with relative frequenc (q_1...q_7) = (4, 5, 7, 8, 10, 12, 20). Now first we draw its equivalent tree.





Optimal Huffman Codes –

Messages Codes

M_1	–	1000
M_2	–	1001
M_3	–	010
M_4	–	011
M_5	–	101
M_6	–	00
M_7	–	11

(g)

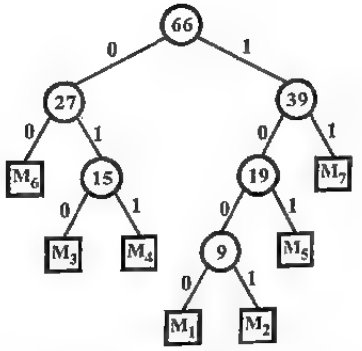
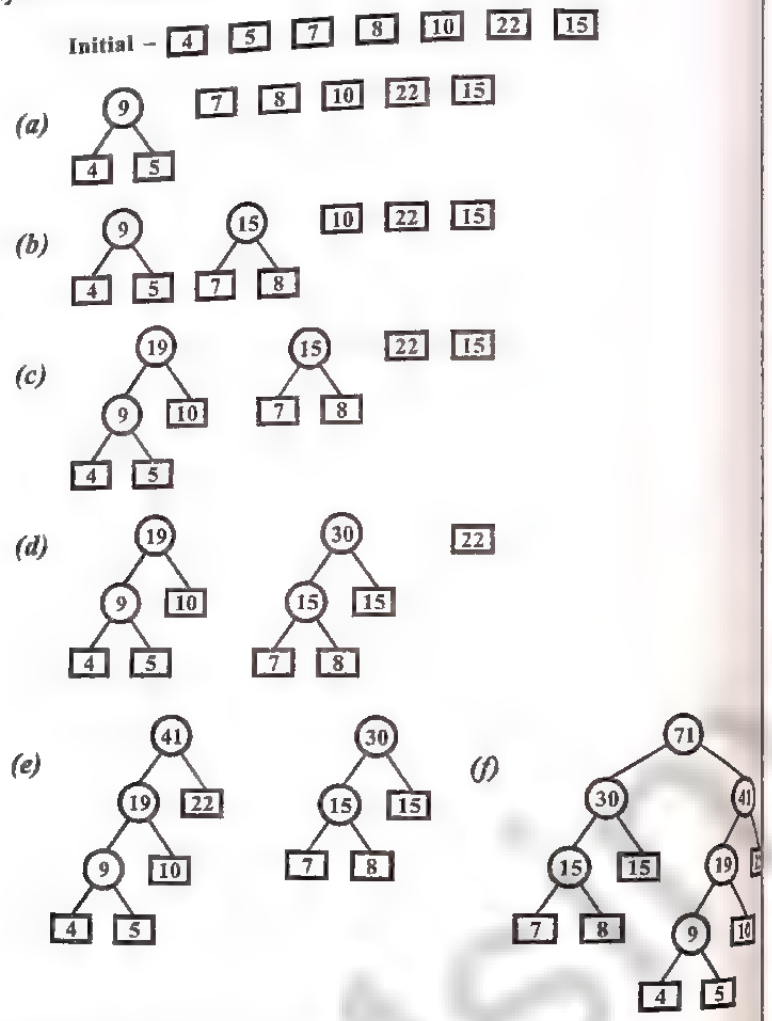


Fig. 2.11

Prob.6. Obtain a set of optimal Huffman codes for the seven messages (M_1.....M_7) with relative frequencies (q_1.....q_7) = (4, 5, 7, 8, 10, 22, 15). Draw the decode tree for this set of codes. ***(R.G.P.V., June 2013)***

Sol. There are seven messages (M_1.....M_7) with relative frequencies (q_1.....q_7) = (4, 5, 7, 8, 10, 22, 15). Now first we draw its equivalent tree.



Thus coded tree is represented as

(g)

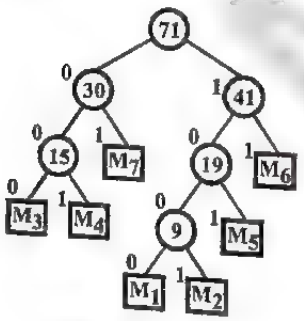


Fig. 2.12

Optimal Huffman Codes –

Messages	Codes
M_1	1000
M_2	1001
M_3	000
M_4	001
M_5	101
M_6	11
M_7	01

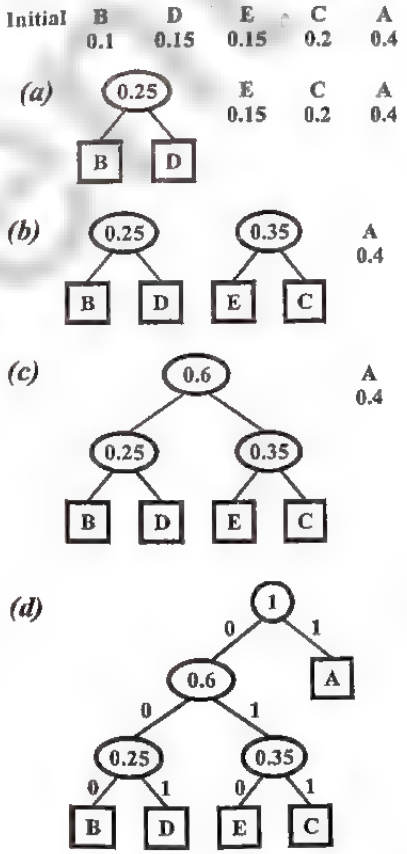
Prob.7. Construct a Huffman code for the following data –

Character	*A*	*B*	*C*	*D*	*E*
Probability	*0.4*	*0.1*	*0.2*	*0.15*	*0.15*

Decode the text whose ending 100010111001010 using the above Huffman code. ***(R.G.P.V., Dec. 2016)***

Sol. There are five characters (A, B, C, D, E) with relative frequencies (0.4, 0.1, 0.2, 0.15, 0.15).

Now, first draw its equivalent tree



Optimal Huffman Codes

A – 1
B – 000
C – 011
D – 001
E – 010

Fig. 2.13

Message Decoding –

$$\underset{A}{1}\ \underset{B}{000}\ \underset{A}{1}\ \underset{C}{011}\ \underset{A}{1}\ \underset{D}{001}\ \underset{E}{010}$$

Hence, the required message is **ABACADE.** **Ans.**

Prob.8. Apply Prim's algorithm to the following graph. Write the complexity. Find the minimum cost.

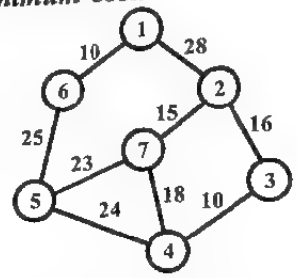


Fig. 2.14 *(R.G.P.V., May 20*

Sol. Prepare the Prim's table to find the minimum cost.

	1	2	3	4	5	6	7
1	0	28	∞	∞	∞	(10)	∞
2	28	0	(16)	∞	∞	∞	15
3	∞	16	0	(10)	∞	∞	∞
4	∞	∞	10	0	24	∞	18
5	∞	∞	∞	24	0	25	(23)
6	10	∞	∞	∞	(25)	0	∞
7	∞	(15)	∞	18	23	∞	0

Here number of nodes are 7. Therefore total number of edge in spanni tree will be 6.

We start a tree that include minimum cost edge is, (1, 6) as shown above table.

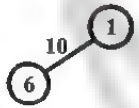


Fig. 2.15 (a)

From node 6 we can choose (6, 1) or (6, 5). Since (6, 1) is already take therefore we take (6, 5).

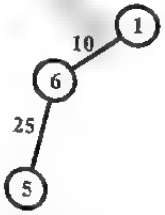


Fig. 2.15 (b)

From node 5 the smallest value is 23, therefore edge (5, 7) will be taken.

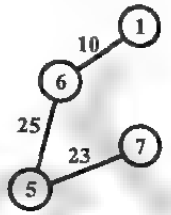


Fig. 2.15 (c)

From node 7 the smallest value is 15, therefore edge (7, 2) will be taken.

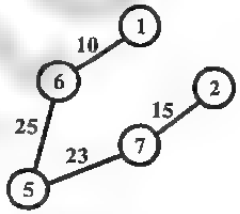


Fig. 2.15 (d)

From node 2 the smallest value is 16, therefore edge (2, 3) will be taken. Since (2, 7) is already taken.

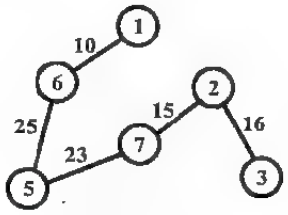


Fig. 2.15 (e)

From node 3 the smallest value is 10, therefore edge (3, 4) will be taken. Hence the minimum cost spanning tree is as follows –

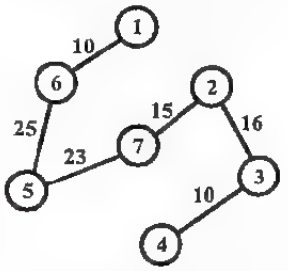


Fig. 2.15 (f)

∴ Minimum cost = 10 + 25 + 23 + 15 + 16 + 10 = 99 **Ans.**

For complexity, refer to Q.12.

Prob.9. Find minimum spanning tree using Prim's algorithm for graph given below –

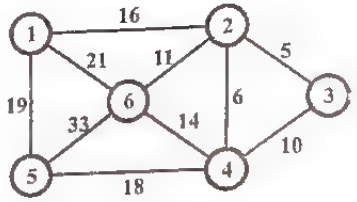


Fig. 2.16 *(R.G.P.V., June 201*

Sol. Fig. 2.17 shows the execution of Prim's algorithm on the graph from fig. 2.16. Minimum spanning tree of the given graph is fig. 2.17 (f).

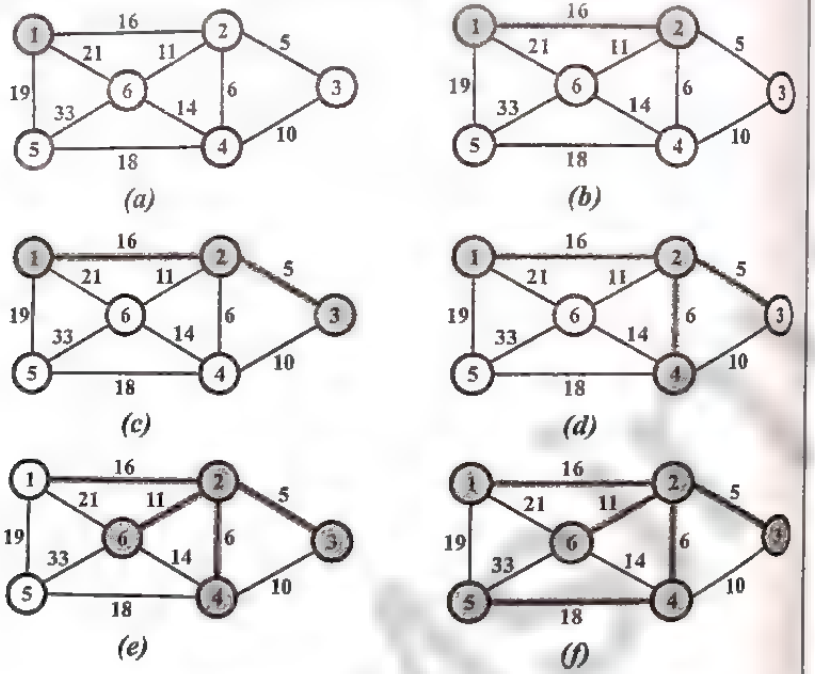


Fig. 2.17

The minimum cost is 56.

Prob.10. What is minimum spanning tree ? Using Prim's algorithm find minimum spanning tree of the following graph –

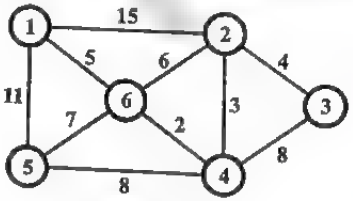


Fig. 2.18 *(R.G.P.V., Dec. 201*

Sol. **Minimum Spanning Tree** – Refer to Q.11.

Problem – Fig. 2.19 shows the execution of Prim's algorithm on the graph from fig. 2.18. The resulting minimum spanning tree is shown in fig. 2.19 (e), having cost 21.

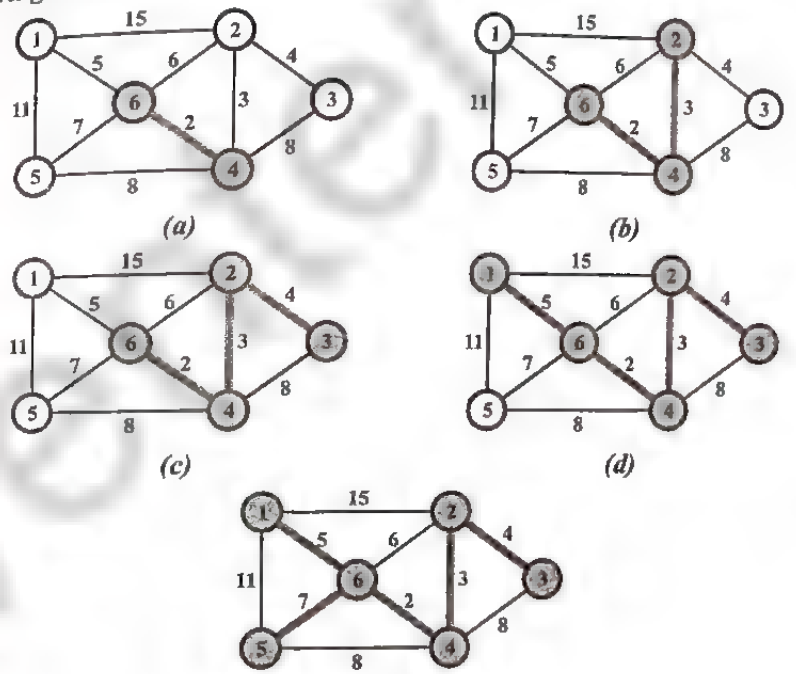


Fig. 2.19

Prob.11. What is greedy strategy ? Write Prim's minimum cost spanning tree algorithm. Determine MST for the following graph.

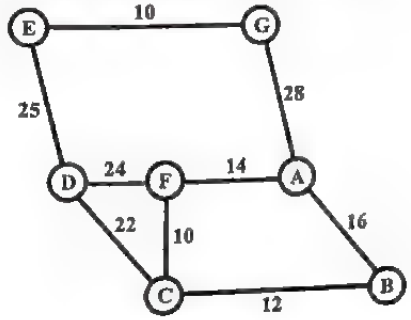


Fig. 2.20

(R.G.P.V., Dec. 2012)

Sol. **Greedy Strategy** – Refer to Q.1.

Prim's Algorithm – Refer to Q.12.

Fig. 2.21 shows the working of Prim's method on the graph of fig. 2.20. The resulting minimum spanning tree is shown in fig. 2.21 (f) have cost 93.

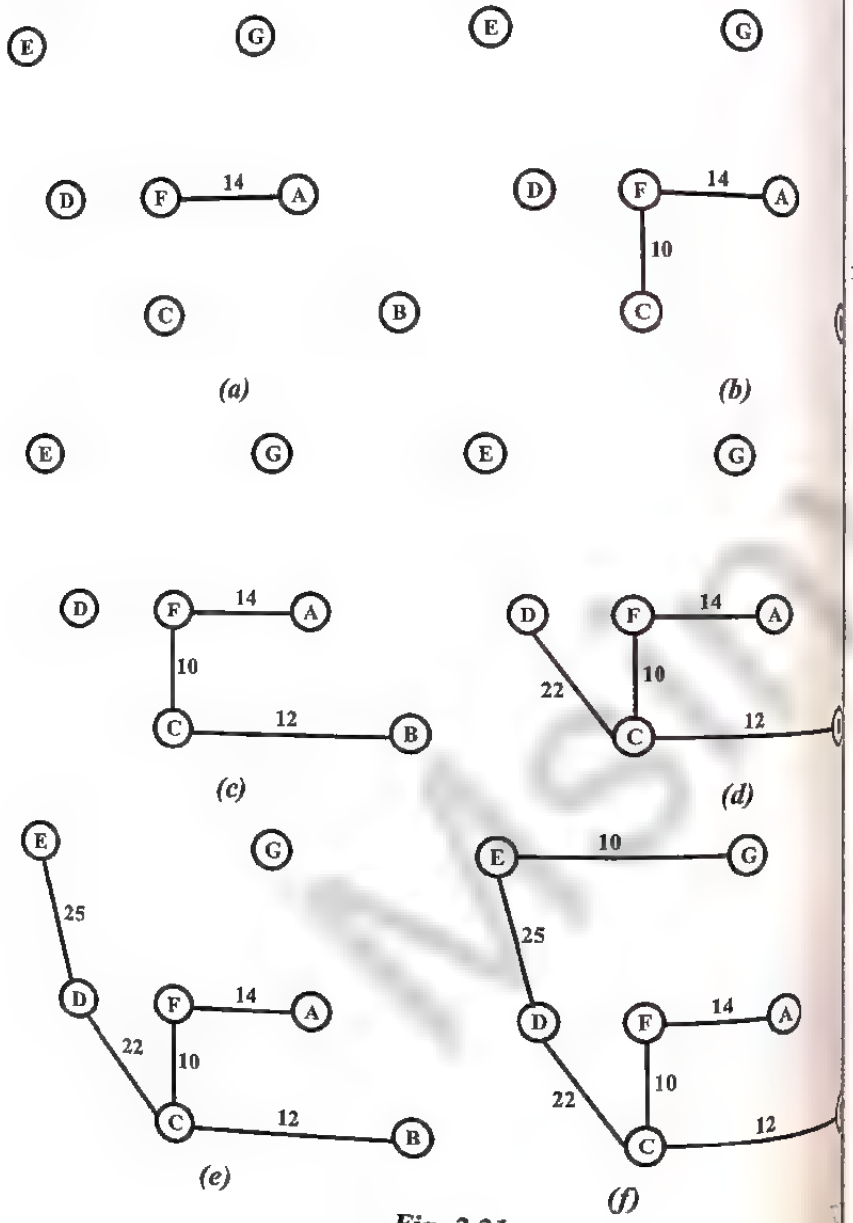


Fig. 2.21

Prob.12. Apply Kruskal's and Prim's algorithm for the following graph. Write their time complexities. Find the minimum cost in each case.

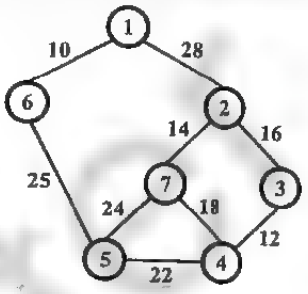


Fig. 2.22 ***(R.G.P.V., May 2019)***

Sol. **Kruskal's Algorithm** – The execution of the Kruskal's algorithm for the given fig. 2.22 is shown in fig. 2.23.

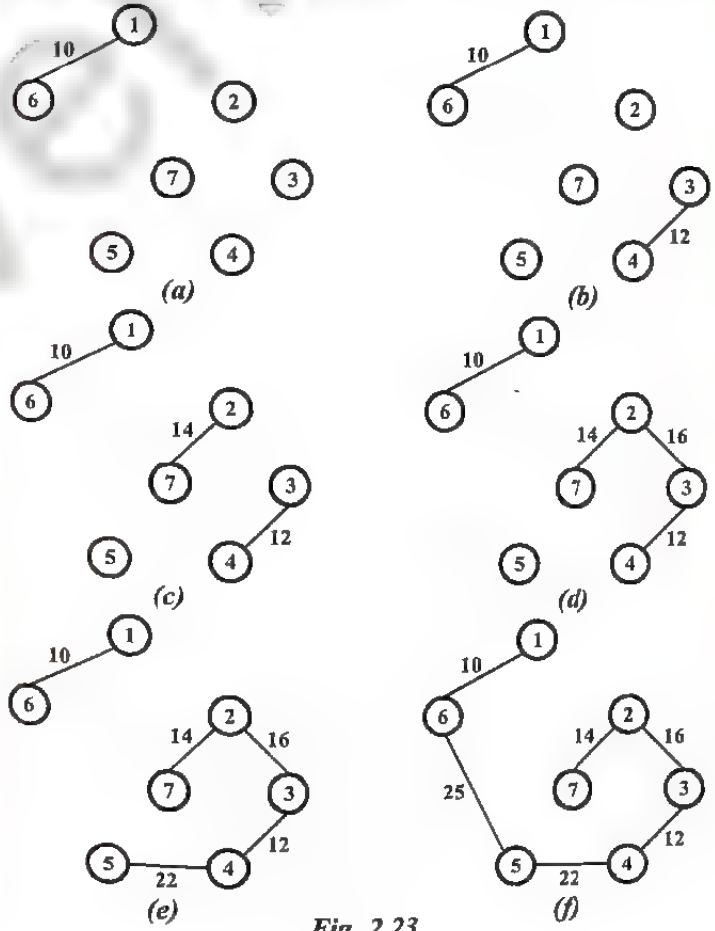


Fig. 2.23

$\therefore$ **Minimum Cost** = 10 + 12 + 14 + 16 + 22 + 25 = 99

Time complexity of Kruskal's algorithm is O(log V)

where V = Number of vertices.

Prim's Algorithm – The execution of the Prim's algorithm for the fig. 2.22 is shown in fig. 2.24.

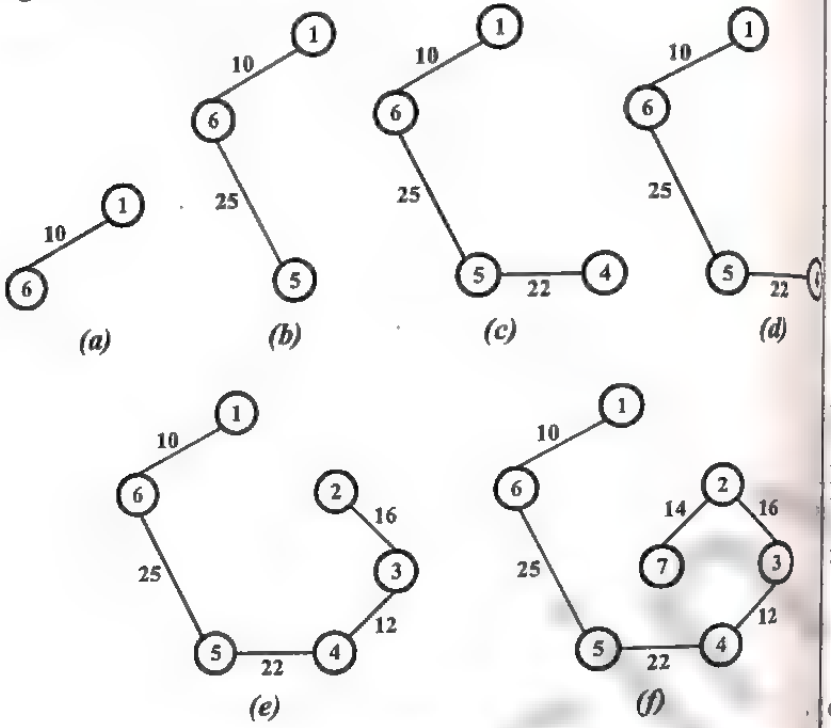


Fig. 2.24

$\therefore$ **Minimum Cost** = 10 + 25 + 22 + 12 + 16 + 14 = 99

Time complexity of Prim's algorithm is $O(V^2)$

Prob.13. Find an optimal solution to the following Knapsack prob

Number of objects n = 3

Knapsack capacity m = 20

Profits $(p_1, p_2, p_3) = (25, 24, 15)$

Weights $(w_1, w_2, w_3) = (18, 15, 10)$.

(R.G.P.V., June

Or

Find an optimal solution to the Knapsack instance n = 3, m = 2

$p_2, p_3) = (25, 24, 15)$ ***and*** $(w_1, w_2, w_3) = (18, 15, 10)$. ***(R.G.P.V., June***

Sol. For solving above Knapsack problem, we use some strategy to determine the fraction of weight which should be included so as to maximize the profit and fill the Knapsack completely.

	(x_1, x_2, x_3)	$\Sigma w_i x_i$	$\Sigma p_i x_i$
(i)	$\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right)$	16.5	24.25
(ii)	$\left(1, \frac{2}{15}, 0\right)$	20	28.2
(iii)	$\left(0, \frac{2}{3}, 1\right)$	20	31
(iv)	$\left(0, 1, \frac{1}{2}\right)$	20	31.5

There are four feasible solutions given above.

(i) At each step, we try to get the maximum profit. The maximum profit we get by object of profit 25 and weight 18. So, we place it in Knapsack first by having $x_1 = 1$. Now, profit is 25, and weight is 18. Total capacity of Knapsack is 20. The next maximum profit we can have 24, having weight 15, but we have only 2 units of weight. So a fraction of this object will be added i.e., $x_2 = \frac{2}{15}$.

Now, fraction of profit that we get is $24 \times \frac{2}{15} = 3.2$.

Total profit gain is 25 + 3.2 = 28.2. And total weight is 20 equal to capacity of Knapsack.

The method used to obtain this solution is termed as greedy method, because at each step (except possibly the last one), we choose to introduce that object which would increase the objective function value the most.

(ii) In above method, although we try to maximize the objective function value, but it does not result into a optimal solution, as weight increases rapidly at each step and we are not able to maximize the profit.

So, in this approach, we include the object in order of nondecreasing weight w_i.

By above method, first we include object of minimum weight, i.e., w_3, so $x_3 = 1$. Its weight is 10 and profit we gain is 15. Now, the total capacity of Knapsack is 20, so we are left with only 10 units of weight. For this, include next minimum weight object w_2 having weight 15 and profit 24. But we have only 10 units, so the fraction we will have is a $x_2 = \frac{10}{15}$ i.e., $x_2 = \frac{2}{3}$.

Now, the profit we gain is $24 \times \frac{2}{3} = 16$.

Total profit gain = 16 + 15 = 31.

Total weight = 20.

(iii) Even after reducing the rate of capacity used at each step, we are not getting an optimal solution. For this, now we try to achieve a balance between the rate at which profit increase and the rate at which capacity used. For this, at each step we include that object which has the maximum profit per unit of capacity used. This means objects are considered in order of ratio p_i/w_i, i.e., the object having maximum profit per unit weight will be used first.

In our example, 1st object has $\frac{\text{profit}}{\text{weight}} = \frac{25}{18} = 1.3$

2nd object has $\frac{\text{profit}}{\text{weight}} = \frac{24}{15} = 1.6$

3rd object has $\frac{\text{profit}}{\text{weight}} = \frac{15}{10} = 1.5$

So, maximum profit is in case of 2nd object, and we include it, $x_2 = 1$. Now, weight is 15 and profit is 24. Further, we add object 3rd next highest profit, but we are left with only 5 units, so fraction of weight used is $\frac{1}{2}$. The weight added is $\frac{10}{2} = 5$ and profit is $\frac{15}{2} = 7.5$. Total profit 24 + 7.5 = 31.5.

This is the best solution, with maximum profit. Greedy Knapsack algorithm 2.5 also uses the same method.

Prob.14. What is Knapsack problem ? Find the optimal solution for Knapsack instance n = 3, m = 20,

(p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).

(R.G.P.V., Dec. 2011,

Sol. Refer to Q.16 and Prob.13.

Prob.15. Find the optimal solution of the Knapsack instance n = 7, m = 15, $(p_1, p_2, \ldots\ldots, p_7)$ = (10, 5, 15, 7, 6, 18, 3) and $(w_1, w_2, \ldots\ldots$ = (2, 3, 5, 7, 1, 4, 1). ***(R.G.P.V., May/June 2006, Dec. 2010,***

Or

Consider n = 7, m = 15, $(P_1, P_2, \ldots\ldots, P_7)$ = (10, 5, 15, 7, 6, 18, 3) and $(W_1, W_2, \ldots\ldots, W_7)$ = (2, 3, 5, 7, 1, 4, 1). Obtain the optimal solution for the Knapsack instance. ***(R.G.P.V., Dec.***

Sol. To solve this problem, we use some strategy to determine the fraction of weight which should be included so as to maximize the profit and fill the Knapsack.

	$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\Sigma w_i x_i$	$\Sigma p_i x_i$
(i)	$\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8}\right)$	5.51	15.76

Now taking maximum profit 18 with weight 4 as –

$x_6 = 1$, $\Sigma w_i x_i < m$.

(ii)	$\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, 1, \frac{1}{8}\right)$	8.51	31.19
(iii)	$\left(\frac{1}{2}, \frac{1}{3}, 1, \frac{1}{5}, \frac{1}{6}, 1, \frac{1}{8}\right)$	12.69	42.44
(iv)	$\left(1, \frac{1}{3}, 1, \frac{1}{5}, \frac{1}{6}, 1, \frac{1}{8}\right)$	13.69	47.44
(v)	$\left(1, \frac{1}{3}, 1, \frac{1}{5}, 1, 1, \frac{1}{8}\right)$	14.52	52.44
(vi)	$\left(1, \frac{1}{3}, 1, \frac{1}{7}, 1, 1, 1\right)$	15	54.67
(vii)	$\left(1, \frac{2}{3}, 1, 0, 1, 1, 1\right)$	15	55.33

At each step, we try to get the maximum profit. The maximum profit we get by step (vii) taking $x_1 = 1$, $x_2 = 2/3$, $x_3 = 1$, $x_4 = 0$, $x_5 = 1$, $x_6 = 1$, and $x_7 = 1$. These fraction of weight provided maximum profit. This is the best solution, with maximum profit.

Prob.16. Consider the Knapsack instance n = 3, (w_1, w_2, w_3) = (2, 3, 4) and (p_1, p_2, p_3) = (1, 2, 5), m = 5. Find the optimal solution.

(R.G.P.V., June 2009, 2013)

Sol. For solving the Knapsack problem, we use some strategy to determine the fraction of weight which should be included so as to maximize the profit and fill the Knapsack completely.

	x_1,	x_2,	x_3	$\Sigma w_i x_i$	$\Sigma p_i x_i$
(i)	$\frac{1}{2}$,	$\frac{1}{3}$,	$\frac{1}{4}$	3	2.42
(ii)	0,	$\frac{1}{3}$,	1	5	5.67
(iii)	1,	1,	0	5	3
(iv)	0,	1,	$\frac{1}{2}$	5	4.5

The maximum profit we get by (ii) taking $x_1 = 0$, $x_2 = 1/3$, $x_3 = 1$. These fraction of weight provided maximum profit is the optimal solution.

Prob.17. A Knapsack capacity is 100. The weights and values of objects are as follows –

Weight w_i :	***10***	***20***	***30***	***40***	***50***
Value p_i :	***20***	***30***	***66***	***20***	***60***

Solve the Knapsack problem using greedy strategy and find maximum profit that can be obtained. ***(R.G.P.V., June***

Sol. We have

$$n = 5$$
$$m = 100$$
$$(p_1, p_2, p_3, p_4, p_5) = (20, 30, 66, 20, 60)$$
$$(w_1, w_2, w_3, w_4, w_5) = (10, 20, 30, 40, 50)$$

First arrange all the elements in the order such that

$$\frac{p[i]}{w[i]} \geq \frac{p[i+1]}{w[i+1]}$$

So,
$$\frac{p[1]}{w[1]} = \frac{20}{10} = 2 \qquad \frac{p[2]}{w[2]} = \frac{30}{20} = 1.5$$
$$\frac{p[3]}{w[3]} = \frac{66}{30} = 2.2 \qquad \frac{p[4]}{w[4]} = \frac{20}{40} = 0.5$$
$$\frac{p[5]}{w[5]} = \frac{60}{50} = 1.2$$

So arrangement will be

$$(p_3, p_1, p_2, p_5, p_4) = (66, 20, 30, 60, 20)$$
$$(w_3, w_1, w_2, w_5, w_4) = (30, 10, 20, 50, 40)$$

Initially, $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 0, 0)$

Now as per the algorithm $U = 100$

Take p_3 and w_3

Check weather $w_3 > U$ (i.e., $30 > 100$)

No, so we continue

$$x[3] = 1.0 \text{ and}$$
$$U = U - w_3 = 100 - 30 = 70$$

Now, take p_1 and w_1

Check weather $w_1 > U$ $(10 > 70)$

No, so we continue

$$x[1] = 1.0 \text{ and}$$
$$U = U - w_1 = 70 - 10 = 60$$

Now, take p_2 and w_2

Check weather $w_2 > U$ (i.e., $20 > 60$)

No, so we continue

$$x[2] = 1.0 \text{ and}$$
$$U = U - w_2 = 60 - 20 = 40$$

Now, take p_5 and w_5

Check weather $w_5 > U$ (i.e., $50 > 40$)

Yes, so we break.

Now clearly there are still some profit and weight left so

$$x[5] = \frac{U}{w_5} = \frac{40}{50} = \frac{4}{5}$$

So, the maximum profit of Knapsack problem by using greedy strategy

x_i $\left(1, 1, 1, 0, \frac{4}{5}\right)$

$\Sigma w_i x_i$ $10 + 20 + 30 + 0 + 50 \times \frac{4}{5} = 100$

$\Sigma p_i x_i$ $20 + 30 + 66 + 0 + 60 \times \frac{4}{5} = 164$

Maximum profit = **164.** **Ans.**

Prob.18. A Knapsack capacity is W = 60. The weights and profits of four items are as follows –

Item	***A***	***B***	***C***	***D***
Profit P_i	***280***	***100***	***120***	***120***
Weight W_i	***40***	***10***	***20***	***24***

Solve the Knapsack problem using greedy strategy and find the maximum profit that can be obtained. ***(R.G.P.V., Nov. 2018)***

Sol. We have

$$n = 4$$
$$W = 60$$
$$(P_1, P_2, P_3, P_4) = (280, 100, 120, 120)$$
$$(W_1, W_2, W_3, W_4) = (40, 10, 20, 24)$$

First arrange all the elements in the order such that

$$\frac{P[i]}{W[i]} \geq \frac{P[i+1]}{W[i+1]}$$

So
$$\frac{P[1]}{W[1]} = \frac{280}{40} = 7$$
$$\frac{P[2]}{W[2]} = \frac{100}{10} = 10$$

$$\frac{P[3]}{W[3]} = \frac{120}{20} = 6$$

$$\frac{P[4]}{W[4]} = \frac{120}{24} = 5$$

So arrangement will be

$(P_2, P_1, P_3, P_4) = (100, 280, 120, 120)$

$(W_2, W_1, W_3, W_4) = (10, 40, 20, 24)$

Initially, $(X_1, X_2, X_3, X_4) = (0, 0, 0, 0)$

Now as per the algorithm $U = 60$ {∴ U=

Take P_2 and W_2

Check weather $W_2 > U$ (i.e. $10 > 60$)

No, so we continue

$X[2] = 1.0$ and

$U = U - W_2 = 60 - 10 = 50$

Now, take P_1 and W_1

Check weather $W_1 > U$ $(40 > 50)$

No, so we continue

$X[1] = 1.0$ and

$U = U - W_1 = 50 - 40 = 10$

Now, take P_3 and W_3

Check weather $W_3 > U$ $(20 > 10)$

Yes, so we break.

Now clearly there are still some profit and weight left so

$$X[3] = \frac{U}{W_3} = \frac{10}{20} = \frac{1}{2}$$

So, the maximum profit of Knapsack problem by using greedy str

$$X_i = \left(1, 1, \frac{1}{2}, 0\right)$$

$$\Sigma W_i X_i = 40 + 10 + 20 \times \frac{1}{2} + 0 = 60$$

$$\Sigma P_i X_i = 280 + 100 + 120 \times \frac{1}{2} + 0 = 440$$

Maximum profit = **440**

Prob.19. There are 5 jobs whose profits $(P_1, \ldots, P_5) = (20, 15, 10, 1, 6)$ and deadlines (2, 2, 1, 3, 3). Find the optimal solution that maximizes profit on scheduling these jobs. Discuss its algorithm too. (R.G.P.V., June 2011, 2013)

Sol.

J	*Assigned Slots*	*Job Considered*	*Action*	*Profit*
ϕ	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1] [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	36

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 36.

Prob.20. Write a greedy algorithm for sequencing unit time jobs with deadlines and profits. Using this algorithm, find the optimal solution when n = 5.

Job	*Profit*	*Deadline*
p_1	20	2
p_2	15	2
p_3	10	1
p_4	5	3
p_5	1	3

(R.G.P.V., Dec. 2016)

Sol. **Algorithm for Job Sequencing Problem** – Refer to Q.22.

Problem –

Here we have, $n = 5$

$(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

So, by using the feasibility rule described in Q.22, we have

J	*Assigned Slots*	*Job Considered*	*Action*	*Profit*
ϕ	None	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit : reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with profit of **40**. **Ans.**

Prob.21. There are 5 jobs whose profits $(P_1, P_2, P_3, P_4, P_5) = (60, 100, 40, 20)$ and deadlines $(D_1, D_2, D_3, D_4, D_5) = (2, 1, 3, 2, 1)$. Find the optimal solution that maximizes profit on scheduling these jobs. (R.G.P.V., Nov. 20

Sol. Here we have, n = 5

$(P_1, P_2, P_3, P_4, P_5) = (60, 100, 20, 40, 20)$

$(D_1, D_2, D_3, D_4, D_5) = (2, 1, 3, 2, 1)$

The given jobs are sorted according to their profit in a descending or Hence, after sorting, the jobs are ordered as shown in below –

Job	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within deadline and contributes maximum profit.

Next, J_1 is selected as it gives more profit as compared to J_4.

In the next clock, J_4 cannot be selected as its deadline is over, hence selected as it executes within its deadline.

The job J_5 is discarded as it cannot be executed within deadline.

Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are be executed within their deadline and gives maximum profit.

Total profit of this sequence is 100 + 60 + 20 = **180.**

Prob.22. Solve the following instances of the single source shortest p problem with vertex 'a' as the source.

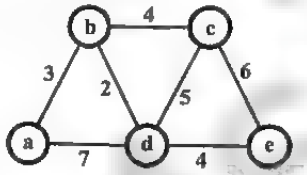


Fig. 2.25 ***(R.G.P.V., Dec. 20***

Sol. The value of distance and the vertices selected at each iteration the for loop for finding all the shortest path from node a is shown in table

Table 2.2 Action of Shortest Path

Iteration	*S*	*Vertex Selected*	*Distance*				
			a	*b*	*c*	*d*	*e*
Initial	–	–	0	3	∞	7	∞
1	{a}	b	0	3	7	5	∞
2	{a, b}	d	0	3	7	5	9
3	{a, b, d}	e	0	3	7	5	9
4	{a, b, d, e}	c	0	3	7	5	9
	{a, b, d, e, c}						

Prob.23. Find the shortest path from vertex 1 to vertex 3 in the following weighted graph using Dijkstra's greedy algorithm.

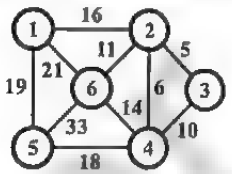
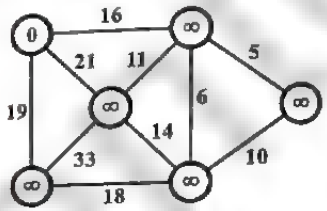
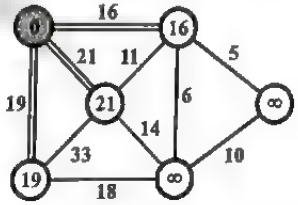


Fig. 2.26 ***(R.G.P.V., Dec. 2014)***

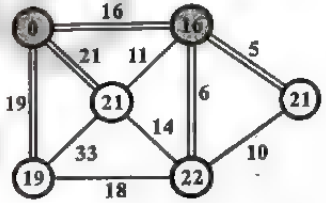
Sol. The execution of Dijkstra's algorithm is shown in fig. 2.27.



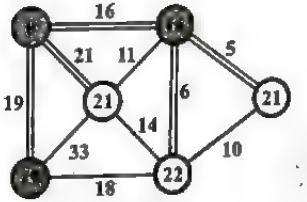
(a)



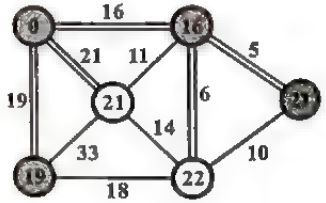
(b)



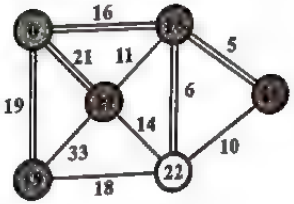
(c)



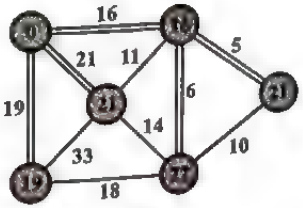
(d)



(e)



(f)



(g)

Fig. 2.27

Prob.24. Write Dijkstra's algorithm to find the shortest path between given vertices. Using this algorithm find shortest path from vertex 1 to vertex 3 in the following weighted graph.

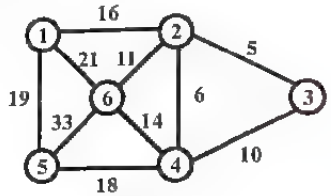


Fig. 2.28

(R.G.P.V., Dec. 20

Sol. Refer to Q.25 and Prob.23.

UNIT 3

CONCEPT OF DYNAMIC PROGRAMMING, PROBLEMS BASED ON THIS APPROACH SUCH AS 0/1 KNAPSACK, MULTISTAGE GRAPH, RELIABILITY DESIGN, FLOYD-WARSHALL ALGORITHM

Q.1. Define dynamic programming. ***(R.G.P.V., June 2015)***

Or

Write short note on dynamic programming.

(R.G.P.V., Dec. 2015, June 2016)

Or

Explain the concept of dynamic programming. (R.G.P.V., Dec. 2016)

Ans. Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

As divide and conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. While, in contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subproblems. In this context, a divide and conquer algorithm does more work than necessary, repeatedly solving the common subproblems.

A dynamic programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered.

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. Dynamic programming is typically applied to optimization problems.

In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Q.2. Give some examples where dynamic programming can be appli

Ans. Dynamic programming can be applied in all the situations where solution to a problem can be viewed as the result of a sequence of decisio

Some of such problems are as follows –

(i) Optimal Merge Pattern – In this pattern, we have to merg pair of files at each step. Here, we have a decision sequence, i.e., determination about the files that should be merged at each step. So we c apply dynamic programming in this case.

(ii) Knapsack Problem – In this problem, we are given a Knapsa with a given capacity m, and some objects each having some given wei and profit.

At each step, we have to decide the values of x_i, $1 \le i \le n$. First we m a decision on x_1 , then on x_2 and so on. Keeping a goal to maximize the pro and $\Sigma w_i x_i \le m$, with a constraint $0 \le x_i \le 1$.

(iii) Shortest Path Problem – In this problem, we try to find the minim distance, i.e., the shortest path between vertex i and j. At each step determine next shortest path. So we apply dynamic programming to above problem.

The main concept behind dynamic programming is principle of optima which states that an optimal sequence of decisions has the property t whatever the initial state and decision are, the remaining decisions must consti an optimal decision sequence with regard to the state resulting from the fi decision.

We apply this on shortest path problem. Here, we assume initially tha $i_1, i_2, \ldots\ldots, i_k, j$ is a shortest path from i to j.That means starting fr position i, we first go to vertex i_1.

Following this decision, new decision is to be made, stating the probl to find to shortest path from vertex i_1 to j. According to principle of optima the sequence i_1, i_2, i_k, j must constitute a shortest i_1 to j path.

If not, let $i_1, r_1, r_2, \ldots\ldots, r_q, j$ be a shortest i_1 to j path.Then i, $r_1, \ldots\ldots, r_q, j$ is an i to j path that is shorter than the path i, $i_1, i_2, \ldots\ldots,$ j.

Therefore, principle of optimality applies to this problem.

Q.3. Explain dynamic programming concept with example.

(R.G.P.V., June 2009, 20

Or

Explain dynamic programming with example.

(R.G.P.V., Dec. 2009, 20

Ans. Refer to Q.1 and Q.2.

Q.4. Write the characteristics of dynamic programming.

(R.G.P.V., Dec. 2014)

Ans. Some characteristics of dynamic programming are as follows –

(i) The problem can be divided into stages, with a policy decision required at each stage.

(ii) Each stage has a number of states associated with the beginning of that stage.

(iii) The effect of the policy decision at each stage is to transform the current state to a state associated with the beginning of the next stage.

(iv) The solution procedure is designed to find an optimal policy for the overall problem.

(v) Given the current state, an optimal policy for the remaining stages is independent of the policy decisions adopted in previous stages. Therefore, the optimal immediate decision depends on only the current state and not on how you got there. This is the principle of optimality for dynamic programming.

(vi) The solution procedure begins by finding the optimal policy for the last stage.

Q.5. What is principle of optimality ? Explain with suitable example.

(R.G.P.V., June 2014, Dec. 2014, 2015)

Or

Explain principle of optimality. ***(R.G.P.V., June 2015)***

Ans. The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Example – Consider a shortest path problem where one way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth, and so on, until vertex j is reached. An optimal sequence is one that results in a path of least length. Assume that i, $i_1, i_2, \ldots\ldots, i_k, j$ is a shortest path from i to j. Starting with the initial vertex i, a decision has been made to go to vertex i_1. Following this decision, the problem state is defined by vertex i_1 and we need to find a path from i_1 to j. It is obvious that the sequence $i_1, i_2, \ldots\ldots, i_k, j$ must form a shortest i_1 to j path. If not, let $i_1, r_1, r_2, \ldots\ldots, r_q, j$ be a shortest $i_1$to j path. Then i, $i_1, r_1, \ldots\ldots, r_q, j$ is an i to j path that is shorter than the path i, $i_1, i_2, \ldots\ldots, i_k, j$. Therefore, the principle of optimality applies for this problem.

Q.6. What is dynamic programming ? Compare it with greedy meth ***(R.G.P.V., Dec. 20***

Ans. Refer to Q.1.

Comparison between greedy method and dynamic programming meth are as follows –

S.No.	Greedy Method	Dynamic Programming
(i)	It first make an optimal choice, without knowing solution to subproblems and then solve remaining subproblem (s).	It solve subproblems first, then use those solution to make an optimal choice.
(ii)	It follows top-down technique.	It follows bottom-up technique.
(iii)	It does not use principle of optimality.	It uses principle of optimality.
(iv)	A greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage a decision is made regarding whether a particular input is an optimal solution.	Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as a result of a sequence of decisions.

Q.7. Discuss the elements of dynamic programming.

Or

What are the key ingredients that an optimization problem must ha for dynamic programming to be applicable ? Explain them.

(R.G.P.V., Dec. 20

Ans. There are two key elements that an optimization problem must ha for dynamic programming to be applicable. They are –

(i) Optimal substructure

(ii) Overlapping subproblems.

(i) Optimal Substructure – The very first step in solving optimization problem by dynamic programming is to characterize the struct of an optimal solution. A problem exhibits or shows ***optimal substructure*** if optimal solution to the problem contains within it optimal solutions to subprobl It is a good clue that dynamic programming can be applied, if a problem exhib optimal substructure. The optimal substructure of a problem often suggest suitable space of subproblems to which dynamic programming can be appl

(ii) Overlapping Subproblems – The second property that optimization problem must have for dynamic programming to be applicable that the space of subproblems must be "small" in the sense that a recurs

algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Generally, the total number of distinct subproblems is a polynomial in the input size. Now when a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has ***overlapping subproblems.*** However, in contrast, a problem for which as divide and conquer approach is suitable usually generates brand-new problems at each step of the recursion. Typically dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution or result in a table where it can be looked up when needed, using constant time per look up.

Memorization is a variant method, for taking advantage of the overlapping-subproblems property. Here the main idea is to ***memorize*** the natural, but inefficient, recursive algorithms.

A memorized recursive algorithm generally maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to show that the entry has yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table. Every time, whenever the subproblem is encountered, the value stored in the table is simply looked up and returned.

Q.8. Give the commonly used designing steps for dynamic programming algorithm. ***(R.G.P.V., Dec. 2014)***

Ans. The development of a dynamic programming algorithm can be broken into a sequence of four steps –

(i) Characterize the structure of an optimal solution.

(ii) Recursively define the value of an optimal solution.

(iii) Compute the value of an optimal solution in a bottom-up fashion.

(iv) Construct an optimal solution from computed information.

Generally steps (i) to (iii) form the basis of a dynamic programming solution to a problem. Step (iv) can be removed if only the value of an optimal solution is required. When we perform step (iv), we maintain additional information during the computation in step (iii) to ease the construction of an optimal solution.

Q.9. Define binomial coefficient. ***(R.G.P.V., June 2015)***

Ans. We use the notation ${}^{n}C_{k}$ (read n choose k) to denote the number of k-combinations of an n-set. The binomial coefficient is given by

$$ {}^{n}C_{k} = \frac{n!}{k!(n-k)!} \quad \text{for } 0 \le k \le n $$

This formula is symmetric in k and n – k –

$$^{n}C_{k} = {}^{n}C_{n-k}$$

These numbers are also known as binomial coefficients, due to their appearance in the binomial expansion –

$$(x + y)^{n} = \sum_{k=0}^{n} {}^{n}C_{k}\, x^{k}\, y^{n-k}$$

Q.10. Give a dynamic programming solution for computing binomial coefficients. ***(R.G.P.V., Dec. 2014)***

Ans. Using dynamic programming approach, binomial coefficient can be calculated recursively by using the relation –

$$^{n}C_{k} = {}^{n-1}C_{k-1} + {}^{n-1}C_{k} \text{ for } n > k > 0$$

and $^{n}C_{0} = {}^{n}C_{n} = 1$

The algorithm 3.1 calculate binomial coefficient using dynamic programming.

Algorithm 3.1

Algorithm Binomial (n, k)

```
for i = 0 to n do
  for i = 0 to min (i, k) do
    if j = = 0 or j = = i then
      C[i, j] = 1
    else
      C[i, j] = C[i – 1, j – 1] + C[i – 1, j]
return C[n, k]
```

Q.11. Discuss 0/1 Knapsack problem. ***(R.G.P.V., June 20***

Ans. 0/1 Knapsack problem is similar to Knapsack problem, we have discussed earlier with slight difference, i.e., here we are given a Knapsack with capacity m, and few objects each with different weight and profit. Here x_i is restricted to have value either 0 or 1, a fraction is not allowed.

We can represent the Knapsack problem KNAP (1, j, y) as –

$$\begin{Bmatrix} \text{Maximize } \sum_{1 \le i \le j} p_i x_i \\ \text{Subject to } \sum_{1 \le i \le j} w_i x_i \le y \\ x_i = 0 \text{ or } 1,\ 1 \le i \le j \end{Bmatrix}$$

We can write Knapsack problem as KNAP (1, n, m), where, n is number of objects, and m is capacity of Knapsack. We have to find the optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$.

Suppose $y_1, y_2, \ldots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$, respectively. If $y_1 = 0$ then $y_2, y_3, \ldots, y_n$ must constitute an optimal sequence for the problem KNAP (2,n,m).

If it does not, then $y_1, y_2, \ldots, y_n$ is not an optimal sequence for the problem KNAP (1, n, m).

On the other hand if $y_1 = 1$, then $y_2, \ldots, y_n$ must constitute an optimal sequence for the problem KNAP (2, n, $m - w_1$). If it does not, then there is another 0/1 sequence

$z_2, z_3, \ldots, z_n$ such that $\sum_{2 \le i \le n} w_i z_i \le m - w_1$ and

$$\sum_{2 \le i \le n} p_i z_i \; \sum_{2 \le i \le n} p_i y_i$$

Hence, the sequence $y_1, z_2, z_3, \ldots, z_n$ is a sequence for equation (i) with greater value. So, here also principle of optimality applies.

Q.12. Show that greedy strategy will not work for 0-1 Knapsack problem. Give a dynamic programming based solution for this problem. ***(R.G.P.V., June 2008, 2013)***

Ans. To see that the greedy strategy does not work for the 0-1 Knapsack problem, consider the problem instance shown in fig. 3.1 (a). There are 3 items, and the Knapsack can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per second of either item 2 or item 3. The greedy strategy, therefore, would take item 1 first. As can be seen from the case analysis in fig. 3.1 (b) however, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.

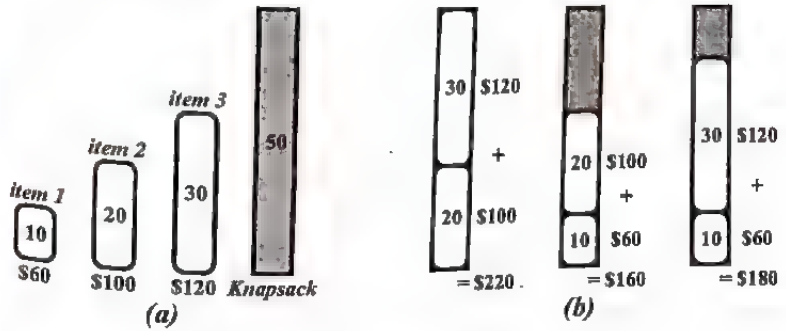


Fig. 3.1

Taking item 1 does not work in the 0-1 problem because the thief unable to fill his Knapsack to capacity and the empty space lowers the effect value per pound of his load. In the 0-1 problem, when we consider an item inclusion in the Knapsack, we must compare the solution to the subproblem which the item is excluded before we can make the choice. The probl formulated in this way gives rise to many overlapping subproblems – a hallm of dynamic programming and indeed, dynamic programming can be used solve the 0-1 problem.

Dynamic Programming Solution to 0/1 Knapsack Problem – Re to Q.11.

Q.13. What is the difference between greedy Knapsack and 0/1 Knapsack Show that 0/1 Knapsack solution not be an optimal solution.

(R.G.P.V., Dec. 2008, June 2009, Dec. 20

Ans. **Difference between Greedy Knapsack and 0/1 Knapsack** – T essential difference between greedy Knapsack and 0/1 Knapsack is that in greedy method only one decision sequence is ever generated. In 0/1 Knapsa many decision sequences may be generated. However, sequences, contain suboptimal subsequences cannot be optimal and so will not be generated.

Greedy Knapsack solves the problem by choosing the item with high ratio at each step and it also allows the fraction of item to maximise the pro Whereas 0/1 Knapsack solves the problem by either selecting each item whole or none.

Proof – Let $f_i(y)$ be the value of an optimal solution to KNAP (1, j Since the principle of optimality holds, we obtain

$$f_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\}$$

For arbitrary $f_i(y)$, $i > 0$, equation (i) generalizes to

$$f_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}$$

Equation (ii) can be solved for $f_n(m)$ by beginning with the knowle $f_0(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then $f_1, f_2, \ldots, f_n$ can be successi computed using equation (ii).

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, 0 $\le m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be comp explicitly. Since each f_i can be computed from f_{i-1} in $\theta(m)$ time, it $\theta(mn)$ time to compute f_n. When the w_i's are real numbers, $f_i(y)$ is needed real numbers y such that $0 \le y \le m$. So f_i cannot be explicitly computed fo y is this range. Even when the w_i's are integer, the explicit $\theta(mn)$ comput of f_n may not be the most efficient computation.

Q.14. Write the recursive equation for 0/1 Knapsack problem based on the principles of optimality. Explain its execution strategy.

(R.G.P.V., May 2019)

Ans. Refer to Q.11, Q.13 and Prob.2.

Q.15. Explain multistage graphs. ***(R.G.P.V., June 2016)***

Or

Write a short note on multistage graphs. ***(R.G.P.V., May 2018)***

Or

What is multistage graph ? ***(R.G.P.V., May 2019)***

Ans. The ***multistage graph problem*** is to find a minimum-cost from source to sink, i.e., target.

A multistage graph is a directed graph with multiple stages. If we denote a graph G = (V, E) in which the vertices are partitioned into $k \ge 2$ disjoint sets, V_i, $1 \le i \le k$.

So that, if there is an edge <u, v> from u to v in E, the $u \in V_i$ and $v \in v_{i+1}$, for some i, $1 \le i < k$. And sets V_1 and V_k are such that $|V_1| = |V_k| = 1$.

In multistage graph, we try to find a minimum-cost path from source to sink.

If we consider s a vertex in V_1 and t a vertex in V_k. The vertex s is supposed as the source and vertex t as sink. The cost of a path from s to t is the sum of the costs of the edges on the path.

Here, each set V_i defines a stage in the graph. Every path starts from stage 1 goes to stage 2 then to stage 3 and so on, because of constraints on E.

A multistage graph is shown in fig. 3.2.

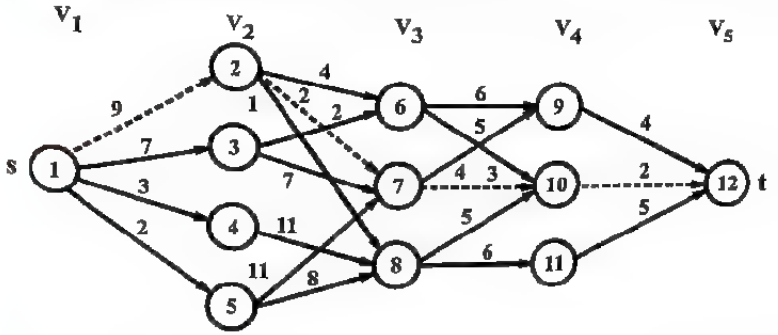


Fig. 3.2 Five Stage Graph

In fig. 3.2, minimum cost s to t path is indicated by dashed line. This method can be used for solving many problems such as allocating some resources to given number of projects with the intend to find the maximum profit.

Q.16. What is multistage graph problem ?Discuss its solution based on dynamic programming approach. Give a suitable algorithm and find its computing time. ***(R.G.P.V., June 2007, 2013, Dec. 201***

Or

Explain multistage graph problem with example. (R.G.P.V., June 201

Or

Define multistage graph problem with the help of suitable algorithm. ***(R.G.P.V., June 201***

Ans. **Multistage Graph Problem** – Refer to Q.15.

Solution Based on Dynamic Programming – A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every source(s) to target (t) path is the result of a sequence of k–2 decisions. The ith decision takes into consideration which vertex in V_{i+1}, $i \le i \le k-2$, is to be on the path. Here principle of optimality holds. Let p (i, j) be a minimum-cost path from vertex j in V_i to vertex t. Let cost (i, j) be the cost of this path. Then, using forward approach, we get

$$\text{cost}(i, j) = \min_{l \in V_{i+1}} \{c(j, l) + \text{cost}(i+1, l)\}$$

$$(j, l) \in E \qquad \ldots(i)$$

As, cost $(k-1, j) = c(j, t)$ if $(j, t) \in E$ and cost $(k-1, j) = \infty$ if $(j, t) \notin E$. The above equation (i) can be solved for cost (1, s) by first computing cost $(k-2, j)$ for all $j \in V_{k-2}$, then cost $(k-3, j)$ for all $j \in V_{k-3}$, and so on and finally cost (1, s). Now we try this method to solve the graph of fig. 3.1 we obtain

cost (3, 6) = min {6 + cost (4, 9), 5 + cost (4, 10)} = 7

cost (3, 7) = min {4 + cost (4, 9), 3 + cost (4, 10)} = 5

cost (3, 8) = 7

cost (2, 2) = min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3, 8)}

cost (2, 3) = 9

cost (2, 4) = 18

cost (2, 5) = 15

cost (1, 1) = min {9 + cost (2, 2), 7 + cost (2, 3),
3 + cost (2, 4), 2 + cost (2, 5)}
= 16

Notice that in calculation of cost (2, 2) we have reused the values of cost (3, 6), cost (3, 7) and cost (3, 8) and so avoided their recomputation. Hence minimum cost s to t path has a cost of 16. This path can be found easily if we record the decision made at each state (vertex). Let d (i, j) be the value of

(where l is a node) that minimizes c (j, l) + cost (i + 1, l). Now for graph shown in fig. 3.2 we obtain

d (3, 6) = 10 ; d (3, 7) = 10 ; d (3, 8) = 10 ;
d (2, 2) = 7 ; d (2, 3) = 6 ; d (2, 4) = 8 ;
d (2, 5) = 8 ; d (1, 1) = 2

To write an algorithm for a general k-stage graph, we impose an ordering on the vertices in V. We require that the n vertices in V are indexed 1 through n. Indices are assigned in order of stages. Here first s, is assigned index 1, then vertices in V_2 are assigned indices, then vertices from V_3, and so on. Vertex t has index n. Thus, indices assigned to vertices in V_{i+1} are bigger than those assigned to vertices in V_i. Thus due to indexing scheme, cost and d can be computed in the order n – 1, n – 2, ..., 1. The first subscript in cost, p and d only identifies the stage number and is omitted in the algorithm 3.2. The resulting algorithm, in pseudocode, is F Graph.

Algorithm 3.2 Multistage Graph Pseudocode Corresponding to the Forward Approach

```
1.  Algorithm F Graph (G, K, n, p)
2.  // The input is a k-stage graph G = (V, E) with n vertices.
3.  // indexed in order of stages. E is a set of edges and
4.  // c[i, j] is the cost of <i, j>, p [i : k] is a minimum-cost path.
5.  {
6.      cost [n] : = 0.0 ;
7.      for j : = n – 1 to 1 step – 1 do
8.      {// Compute cost [j].
9.      Let r be a vertex such that <j, r> is an edge
10.     of G and c [j, r] + cost [r] is minimum ;
11.     cost [j] : = c [j, r] + cost [r] ;
12.     d [j] : = r ;
13. }
14. // Find a minimum-cost path
15. p[1] : = 1 ; p [k] : = n ;
16. for j : = 2 to k – 1 do p [j] : = d [p [j – 1]] ;
17. }
```

The multistage graph problem can also be solved by using the backward approach. Let bp (i, j) be a minimum-cost path from vertex s to a vertex j in V_i. Let b cost (i, j) be the cost of bp (i, j). From the backward approach, we get

$$b\ cost\ (i, j) = \min_{l \in V_{i-1}} \{b\ cost\ (i-1, l) + c(l, j)\} \quad \ldots(i)$$

$$(l, j) \in E$$

Since b cost (2, j) = c (1, j) if <1, j> ∈ E and b cost (2, j) = ∞ if <i, j> ∉ E, b cost (i, j) can be computed using equation (ii) by first computing b cost for = 3, then for i = 4 and so on. For the previous graph shown in fig. 3.2, we obtain

b cost (3, 6) = min {b cost (2, 2) + c (2, 6), b cost (2, 3) + c (3, 6)}
= min {9 + 4, 7 + 2} = 9

b cost (3, 7) = 11
b cost (3, 8) = 10
b cost (4, 9) = 15
b cost (4, 10) = 14
b cost (4, 11) = 16
b cost (5, 12) = 16

Time Complexity – FGraph algorithms complexity analysis is fairl straight forward. If G is represented by its adjacency lists, then r in line 9 of algorithm 3.2 can be found in time proportional to the degree of vertex Thus, if G has $|E|$ edges, then the time for the **for** loop of line 7 is $\Theta(|V|+|E|)$ The time for the **for** loop of line 16 is $\Theta(k)$. Thus, the total time is $\Theta(|V|+|E|)$

Q.17. What is reliability design problem ? Explain.
(R.G.P.V., Dec. 2009, 201

Or

Explain how a reliability design can be obtained using dynami programming. ***(R.G.P.V., Dec. 201***

Or

How reliability of a system is calculated ? ***(R.G.P.V., June 201***

Or

Write short note on reliability design. (R.G.P.V., Dec. 2011, June 201

Or

What is the concept of reliability design in dynamic programming ?
(R.G.P.V., Dec. 201

Or

Explain reliability design problem with suitable example.
(R.G.P.V., Dec. 201

Ans. In reliability design, the problem is to design a system that composed of several devices connected in series as shown in fig. 3.3.

Fig. 3.3 n Devices D_i, $1 \le i \le n$, Connected in Series

If we imagine r_i as the reliability of a device (i.e. probability that a device D_i will work properly).

Then the reliability of the function can be given as $\prod_{r_i}$.

If $r_i = 0.99$ and n = 10 that n devices are set in a series, $1 \le i \le 10$, then reliability of whole system $\prod_{r_i}$ can be given as –

$$\prod_{r_i} = 0.904$$

So, if we duplicate the devices at each stage then reliability of the system can be increased.

Say, multiple copies of same device type are connected in parallel through the use of switching circuits. This is shown in fig. 3.4.

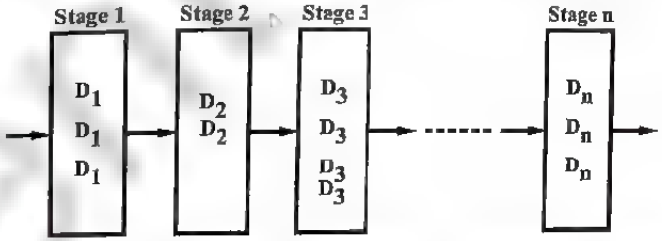


Fig. 3.4 Multiple Devices Connected in Parallel in Each Stage

Here, switching circuit determines which devices in any given group are functioning properly. Then they make use of such devices at each stage, that result is increase in reliability at each stage. If at each stage, there are m_i similar type of devices D_i, then the probability that all m_i have a malfunction is $(1-r_i)^{m_i}$, which is very very less.

And reliability of stage i becomes $(1-(1-r_i)^{m_i})$. Thus if $r_i = 0.99$ and $m_i = 2$, then the stage reliability becomes 0.9999 which is almost equal to 1. Which is much better than that of previous case or we can say the reliability is little less than $1-(1-r_i)^{m_i}$ because of less reliability of switching circuits.

In reliability design, we try to use device duplication to maximize reliability. But this maximization should be considered along with cost.

Let c is the maximum allowable cost and c_i be the cost of each unit of device i.

Then the maximization problem can be given as follows –

Maximize $\prod_{1 \le i \le n} \phi_i(m_i)$

Subject to $\sum_{1 \le i \le n} c_i m_i \le c$

$m_i \ge 1$ and integer $1 \le i \le n$.

Here, ϕ_i (m_i) denotes the reliability of stage i.

The reliability of system can be given as –

$\Pi_{1 \le i \le n}\ \phi_i\ (m_i)$.

If we increase, number of devices at any stage beyond the certain limit then also only cost will increase but reliability could not increase.

Q.18. Briefly explain the concept of dynamic programming. Discuss how a reliability design can be obtained using dynamic programming. ***(R.G.P.V., Nov. 2018)***

Ans. Refer to Q.1 and Q.17.

Q.19. What is all-pair shortest path problem ?

Or

Write an algorithm to find all-pair shortest path. Derive its complexity. ***(R.G.P.V., Dec. 2010)***

Ans. Let G = (V, E) be a directed graph with n vertices. Let cost be a cost adjacency matrix for G such that cost (i, i) = 0, $1 \le i \le n$. Now cost (i, j) is the length (or cost) of edge <i, j> if <i, j> $\in$ E(G) and cost (i, j) = ∞ if $i \ne j$ and <i, j> $\notin$ E(G). The ***all-pairs shortest-path problem*** is to find a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving or calculating n single-source problems using the algorithm to shortest paths shown below –

Algorithm 3.3 Function to Compute Lengths of Shortest Paths

```
0   Algorithm AllPaths (cost, A, n)
1   // cost [1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices ; A [i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost [i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i : = 1 to n do
6           for  j : = 1 to n do
7               A [i, j] : = cost [i, j] ; // Copy cost into A
8       for k : = 1 to n do
9           for i : = 1 to n do
10              for j : = 1 to n do
11                  A [i, j] : = min (A [i, j], A[i, k] + A[k, j]) ;
12  }
```

Complexity – Here each application of the procedure given in algorithm 3.4 needs $O(n^2)$ time. Thus the matrix A can be obtained in $O(n^3)$ time.

Q.20. Define all-pairs shortest path problem. Discuss solution of the problem based on dynamic programming. ***(R.G.P.V., Dec. 2007)***

Ans. Refer to Q.19.

Here the main restriction is that G have no cycles with negative length. Notice that if we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\infty$.

Let us consider a shortest i to j path in G, $i \ne j$. This path starts at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. Here we assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length (no cycles has negative length). Now if k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. Hence the principle of optimality holds. This indicates us to use dynamic programming approach.

If k is the intermediate vertex with highest index, then the i to k path is a shortest i to k path in G going through no vertex with index greater than k–1. Here similarly the k to j path is a shortest k to j path in G going through no vertex of index greater than k–1. We can regard the construction of a shortest i to j path as first requiring a decision as to which is the highest indexed intermediate vertex k. After this decision has been made, we have to determine two shortest paths, one from i to k and the other from k to j. Neither of these may go through a vertex with index greater than k–1. Here $A^k(i, j)$ is used to represent the length of a shortest path from i to j going through no vertex of index greater than k, we get

$$A(i, j) = \min_{1 \le k \le n} \{\min \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, \text{cost}(i, j)\}$$

Clearly, $A^0(i, j) = \text{cost}(i, j)$, $1 \le i \le n$, $1 \le j \le n$. A recurrence for $A^k(i, j)$ is obtained by using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$. If it does not, then no intermediate vertex has index greater than k–1. Hence $A^k(i, j) = A^{k-1}(i, j)$. Combining this we obtain.

$$A^k(i, j) = \min \{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\},\ k \ge 1.$$

Example – Fig. 3.5 shows a digraph with its matrix A^0. For this graph $A^2(1, 3) \ne \min \{A^1(1, 3), A^1(1, 2) + A^1(2, 3) = 2$. Here instead of seeing that $A^2(1, 3) = -\infty$. The length of the path.

1, 2, 1, 2, 1, 2, ... , 1, 2, 3

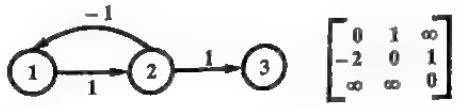


Fig. 3.5 Graph with Negative Cycle

can be made arbitrarily small. This is due to the presence of cycle 1 2 1 which has a length of – 1.

Q.21. Explain Floyd-Warshall algorithm with the help of an example.
(R.G.P.V., Dec. 2013)

Or

Explain Floyd-Warshall algorithm with suitable example.
(R.G.P.V., June 2014)

Or

Give Floyd-Warshall algorithm. What is its running time ?
(R.G.P.V., Dec. 2011, 2012)

Or

Write Floyd-Warshall algorithm to solve all-pair shortest path problem. Also write its complexity.
(R.G.P.V., June 2009, 2013)

Or

Explain Warshalls algorithm. ***(R.G.P.V., June 2015)***

Or

Discuss Floyd-Warshall algorithm. Write its pseudocode.
(R.G.P.V., June 2016)

Or

Explain how to implement Warshall's algorithm without using extra memory for storing elements of the algorithms intermediate matrices.
(R.G.P.V., May 2018)

Or

Write down the pseudocode for Floyd-Warshall algorithm. Take one graph and apply this algorithm to find all pair shortest path on it.
(R.G.P.V., May 2018)

Ans. In the Floyd-Warshall algorithm, we use a different characterization of the structure of a shortest path than we used in the matrix-multiplication based all pairs algorithms. The algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$. The Floyd-Warshall algorithm is based on the following observation. Suppose the vertices of G be $V = \{1, 2, \ldots, n\}$, and assume a subset $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex

i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. It thus has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall (W) algorithm

```
1  n ← rows [W]
2  D(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j = 1 to n
6                 d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D(n)
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3 – 6. Each execution of line 6 takes O(1) time. Thus the algorithm runs in time $O(n^3)$.

In the Floyd-Warshall algorithm, there are many ways for constructing shortest paths. One way is to compute the matrix D of shortest path weights and then construct the predecessor matrix π from the D matrix. This method can be implemented to run in $O(n^3)$ time. Given the predecessor matrix π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure can be used to print the vertices on a given shortest path. We can compute the predecessor matrix π online just as the Floyd-Warshall algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices $\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(n)}$, where $\pi = \pi^{(n)}$ and $\pi_{ij}^{(k)}$ is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

We can give a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all. Thus

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

Formally, for $k \geq 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}.$$

Example – Refer to Prob.9.

Q.22. What is LCS problem ? Write an algorithm to solve longest common sequence problem using dynamic programming approach.
(R.G.P.V., Dec. 2011)

Ans. **LCS Problem** – Given a sequence X of symbols, a subsequence of X is defined to be any contiguous portion of X. For instance if $X = x_1, x_2, x_4, x_5, x_2, x_3$ and x_1, x_2, x_3 are subsequences of X. Given two sequences X and Y, present an algorithm that will identify the longest subsequence that is common to X and Y. This problem is known as the longest common subsequence problem.

Algorithm – The b table returned by LCS – LENGTH can be used to quickly construct an LCS of $X = (x_1, x_2, \ldots\ldots, x_m)$ and $Y = (y_1, y_2, \ldots\ldots, y_n)$. We just start at b [m, n] and trace through the table following the arrows. Whenever we encounter a "↖" in entry b[i, j], it implies that $x_i = y_j$ is an element of the LCS. The elements of the LCS are encountered in reverse order by this method. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial invocation is PRINT-LCS (b, X, length [X], length [Y]).

```
PRINT-LCS (b, X, i, j)
1  if i = 0 or j = 0
2  then return
3  if b [i, j] = "↖"
4      then PRINT-LCS (b, X, i – 1, j – 1)
5           print x_i
6  else if b [i, j] = "↑"
7.     then PRINT-LCS (b, X, i – 1, j)
8.  else PRINT-LCS (b, X, i, j – 1)
```

NUMERICAL PROBLEMS

Prob.1. Solve the following instance of 0/1 Knapsack problem using dynamic algorithm.

Item	*1*	*2*	*3*	*4*
Weight	*4*	*7*	*5*	*3*
Value	*$40*	*$42*	*$25*	*$12*

The capacity of Knapsack m is 10. ***(R.G.P.V., Dec. 20***

Sol. We have n = 4

m = 10

$(v_1, v_2, v_3, v_4) = (40, 42, 25, 12)$

$(w_1, w_2, w_3, w_4) = (4, 7, 5, 3)$

$S^0 = \{(0, 0)\}$ $\quad S_1^0 = \{(40, 4)\}$

$S^1 \{(0, 0), (40, 4)\}$ $\quad S_1^1 = \{(42, 7), (82, 11)\}$

$S^2 = \{(0, 0), (40, 4), (42, 7), (82, 11)\}$

Purging (v_j, w_j) with $w_j > m$

$S^2 = \{(0, 0), (40, 4), (42, 7)\}$

$S_1^2 = \{(25, 5), (65, 9), (67, 12), (107, 16)\}$

$S^3 = \{(0, 0), (40, 4), (42, 7), (25, 5), (65, 9), (67, 12), (107, 16)\}$

Purging (v_j, w_j) with $w_j > m$.

$S^3 = \{(0, 0), (40, 4), (42, 7), (25, 5), (65, 9)\}$

Dominance rule i.e.

If $v_j \le v_k$ and $w_j \ge w_k$ then discard (v_j, w_j)

So, applying dominance rule on (40, 4) and (25, 5) we have,

$S^3 = \{(0, 0), (40, 4), (42, 7), (65, 9)\}$

$S_1^3 = \{(12, 3), (52, 7), (54, 10), (77, 12)\}$

$S^4 = \{(0, 0), (12, 3), (40, 4), (42, 7), (52, 7), (54, 10), (65, 9), (77, 12)\}$

Purging (v_j, w_j) with $w_j > m$ and applying dominance rule, we have

$S^4 = \{(0, 0), (12, 3), (40, 4), (52, 7), (65, 9)\}$

Hence, the maximum profit = 65 and the value of x i.e. $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ **Ans.**

Prob.2. Define how Knapsack problem is solved by using dynamic programming approach. Consider –

$n = 3$, $(w_1, w_2, w_3) = (2, 3, 3)$

$(p_1, p_2, p_3) = (1, 2, 4)$ and $m = 6$, find optimal solution for the given data.

(R.G.P.V., Dec. 2008, 2009, June 2010, 2011, Dec. 2015)

Sol. **Solution of Knapsack Problem** – Refer to Q.11.

Problem – Given $n = 3$, $(w_1, w_2, w_3) = (2, 3, 3)$

$(p_1, p_2, p_3) = (1, 2, 4)$ and $m = 6$

For these data we have

$S^0 = \{(0, 0)\}$; $S_1^0 = \{(1, 2)\}$

$S^1 = \{(0, 0), (1, 2)\}$; $S_1^1 = \{(2, 3), (3, 5)\}$

$S^2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$; $S_1^2 = \{(4,3), (5, 5), (6, 6), (7, 8)\}$

$S^3 = \{(0, 0), (1,2), (4, 3), (5, 5), (6, 6), (7, 8)\}$

The pair (3, 5) and (2, 3) has been eliminated from S^3 as a result of the dominance rule.

Purging (p_j, w_j) with $w_j > m$

$S^3 = \{(0, 0), (1, 2), (4, 3), (5, 5), (6, 6)\}$

With m = 6, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3. The tup
(6, 6) $\notin S^2$ and so we must set $x_3 = 1$. The pair (6, 6) came from the pa
$(6 - p_3, 6 - w_3) = (2, 3)$. Hence, $(2, 3) \in S^2$. Since $(2, 3) \notin S^1$, we can s
$x_2 = 1$. The pair (2, 3) came from the pair $(2 - p_2, 3 - w_2) = (0, 0)$. Hence
$0) \in S^1$. Since $(0, 0) \in S^0$, we obtain $x_1 = 0$. Hence an optimal solution is $(x_1$
$x_2, x_3) = (0, 1, 1)$.

***Prob.3. Consider the Knapsack instance with 5 objects and a capac
m = 11, profits p = (5, 4, 7, 2, 3) and weights w = (4, 3, 6, 2, 2). Solv
using a dynamic programming approach.*** ***(R.G.P.V., Dec. 201***

Sol. We have n = 5

m = 11

$(p_1, p_2, p_3, p_4, p_5) = (5, 4, 7, 2, 3)$

$(w_1, w_2, w_3, w_4, w_5) = (4, 3, 6, 2, 2)$

So, by using dynamic programming

$S^0 = \{(0, 0)\}$; $S_1^0 = \{(5, 4)\}$

$S^1 = \{(0, 0), (5, 4)\}$; $S_1^1 = \{(4, 3), (9, 7)\}$

$S^2 = \{(0, 0), (4, 3), (5, 4), (9, 7)\}$

$S_1^2 = \{(7, 6), (12, 10), (11, 9), (16, 13)\}$

$S^3 = \{(0, 0), (4, 3), (5, 4), (7, 6), (9, 7), (12, 10), (11, 9), (16, 1$

Purging $(p_j\ w_j)$ with $w_j > m$

$S^3 = \{(0, 0), (4, 3), (5, 4), (7, 6), (9, 7), (12, 10), (11, 9)\}$

$S_1^3 = \{(2, 2), (6, 5), (7, 6), (9, 8), (11, 9), (13, 11), (14, 12)\}$

$S^4 = \{(0, 0), (2, 2), (4, 3), (5, 4), (6, 5), (7, 6), (7, 6), (9, 7), (9,$
$(11, 9), (11, 9), (12, 10), (13, 11), (14,$

Purging (p_j, w_j) with $w_j > m$ and applying dominance rule
i.e., if $p_j \le p_k$ and $w_j \ge w_k$, then remove (p_j, w_j)
So,

$S^4 = \{(0, 0), (2, 2), (4, 3), (5, 4), (6, 5), (7, 6), (9, 7), (11, 9),$
$(12, 10), (13,$

$S_1^4 = \{(3, 2), (6, 4), (7, 5), (8, 6), (9, 7), (10, 8), (12, 9), (14,$
$(15, 12), (16,$

Applying dominance rule one purging

$S^5 = \{(0, 0), (3, 2), (4, 3), (6, 4), (7, 5), (8, 6), (9, 7), (10, 8),$
$(12, 9), (14,$

Hence, the solution of 0/1 Knapsack is 14 profit on total weight 11 a
can be found by value of $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 1, 1)$ or $(0, 1, 1, 0,$

Prob.4. Find optimal solution for 0/1 Knapsack problem $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$, $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$ and m = 30.
(R.G.P.V., May/June 2006, June 2014)
Or

Find the optimal solution for 0/1 Knapsack problem $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$, $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$ and w = 30. (R.G.P.V., Nov. 2018)

Sol. n = 4

m = 30

$(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$

$(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$

$S^0 = \{(0, 0)\}$; $S_1^0 = \{(2, 10)\}$

$S^1 = \{(0, 0), (2, 10)\}$; $S_1^1 = \{(5, 15)\ (7, 25)\}$

$S^2 = \{(0, 0)\ (2, 10)\ (5, 15)\ (7, 25)\}$

$S_1^2 = \{(8, 6)\ (10, 16)\ (13, 21)\ (15, 31)\}$

$S^3 = \{(0, 0)\ (2, 10)\ (5, 15)\ (7, 25)\ (8, 6)\ (10, 16)\ (13, 21)\ (15, 31)\}$

Using dominance rules, we have

$S^3 = \{(0, 0)\ (8, 6)\ (10, 16)\ (13, 21)\ (15, 31)\}$

$S_1^3 = \{(1, 9)\ (9, 15)\ (11, 25)\ (14, 30)\ (16, 40)\}$

$S^4 = \{(0, 0)\ (8, 6)\ (1, 9)\ (9, 15)\ (11, 25)\ (14, 30)\ (16, 40)\}$

Purging $(p_j\ \ w_j)$ with $w_j > m$

$S^4 = \{(0, 0)\ (8, 6)\ (1, 9)\ (9, 15)\ (11, 25)\ (14, 30)\}$

With m = 30 the value of $f_4(30)$ is given by the tuple (14, 30) in S^4. The tuple (14, 30) $\notin S^3$ and so we must set $x_4 = 1$. The pair (14, 30) came from the pair $(14 - p_n, 30 - w_n) = (13, 21) \in S^3$. Since $(13, 21) \notin S^2$, we can set $x_3 = 1$. But it came from the $(13 - p_{n-1}, 21 - w_{n-1}) = (5, 15) \in S^2$ since $(5, 15) \notin S^1$. We can set $x_2 = 1$. But (5, 15) came from $(5 - p_{n-2}, 15 - w_{n-2}) = (0, 0)$. Hence $(0, 0) \in S^1$, since $(0, 0) \in S^0$ then we obtain $x_1 = 0$.

Hence an optimal solution is $(x_1, x_2, x_3, x_4) = (0, 1, 1, 1)$.

Prob.5. Find a minimum cost path from 's' to 't' in multistage graph using dynamic programming –

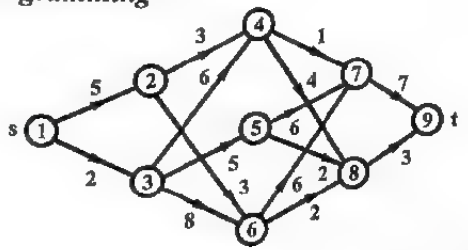


Fig. 3.6 ***(R.G.P.V., May/June 2006)***

Sol. **Solution Based on Dynamic Programming** – A dynamic programm formulation for a K-stage graph problem is obtained by first noticing that e source (s) to target (t) path is the result of a sequence of k – 2 decisions.

Let p (i, j) be a minimum cost path from vertex j in V_1 to vertex t cost (i, j) be the cost of this path. Then, using forward approach, we g

$$\text{cost}(i, j) = \min_{\substack{l \in V_{i+1} \\ (j, l) \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

As, cost (k – 1, j) = C(j, t) if (j, t) ∈ E and cost (k – 1, j) = ∞ if (j, t

We first compute cost (k – 2, j) for all $j \in V_{k-2}$

$$\text{cost}(3, 4) = \min\{1 + \text{cost}(4, 7), 4 + \text{cost}(4, 8)\}$$
$$= \min\{8, 7\} = 7$$
$$\text{cost}(3, 5) = \min\{6 + \text{cost}(4, 7), 2 + \text{cost}(4, 8)\} = 5$$
$$\text{cost}(3, 6) = \min\{6 + \text{cost}(4, 7), 2 + \text{cost}(4, 8)\} = 5$$
$$\text{cost}(2, 2) = \min\{3 + \text{cost}(3, 4), 3 + \text{cost}(3, 6)\} = 8$$
$$\text{cost}(2, 3) = 10$$
$$\text{cost}(1, 1) = \min\{5 + \text{cost}(2, 2), 2 + \text{cost}(2, 3)\}$$
$$= \min\{5 + 8, 2 + 10\} = 12.$$

Notice that in calculation of cost (2, 2) we have reused the value (3, 4) and (3, 6) and avoided their computation. Hence minimum cost p path has a cost of 16. This path can be found easily if we record the deci made at each state. The minimum path from s to t is as –

Cost of Path = **12**

Prob.6. Find a minimum cost path from 's' to 't' in multistage g using dynamic programming –

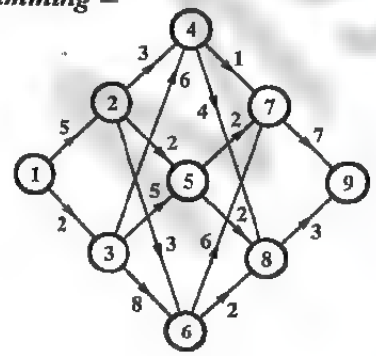


Fig. 3.7 ***(R.G.P.V., June***

Sol. Similar as Prob.5.

Prob.7. Find minimum cost path from 's' to 't' in multistage graph using dynamic programming.

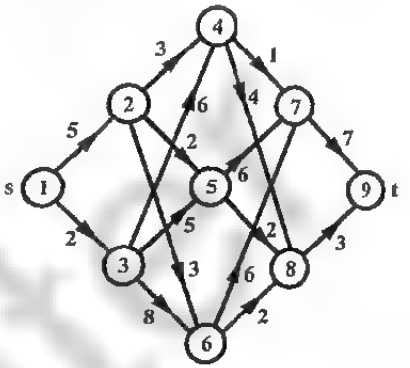


Fig. 3.8 ***(R.G.P.V., Dec. 2015)***

Sol. Similar as Prob.5.

Prob.8. Solve the following multistage problem using both forward and backward reasoning. ***(R.G.P.V., June 2017)***

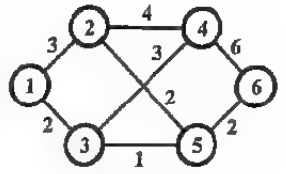


Fig. 3.9

Sol. (i) Solution using Forward Approach –

cost (i, j) = min{c(j, *l*) + cost(i + 1, *l*)}

Here, j = Current vertex

i = Group number in which vertex is placed

c(j, *l*) = Weight of path going from j to *l*

l = Next vertex which is connected to j.

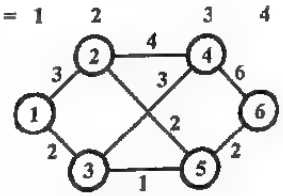


Fig. 3.10

So, in forward approach we start from Sink i.e., vertex 6

We know that sink always have cost = 0 in forward approach.

So, cost(4, 6) = 0

Now

$$\text{cost}(3, 4) = \min\{c(4, 6) + \text{cost}(4, 6)\}$$
$$= \min\{6\}$$
$$= 6$$
$$\text{cost}(3, 5) = 2$$

Now

$$\text{cost}(2, 2) = \min\{c(2, 4) + \text{cost}(3, 4), C(2, 5) + \text{cost}(3, 5)\}$$
$$= \min\{4 + 6, 2 + 2\}$$
$$= \min\{10, 4\}$$
$$= 4$$

$$\text{cost}(2, 3) = \min\{c(3, 4) + \text{cost}(3, 4), C(3, 5) + \text{cost}(3,$$
$$= \min\{3 + 6, 1 + 2\}$$
$$= \min\{9, 3\}$$
$$= 3$$
$$\text{cost}(1, 1) = \min\{c(1, 2) + \text{cost}(2, 2), C(1, 3) + \text{cost}(2,$$
$$= \min\{3 + 4, 2 + 3\}$$
$$= \min\{7, 5\}$$
$$= 5$$

Hence, the minimum cost = 5

and the path will be

$$1 \rightarrow 3 \rightarrow 5 \rightarrow 6$$

(ii) Solution using Backward Approach –

Formula –

$$b\ cost(i, j) = \min\{b\ cost(i - 1, l) + c(l, j)\}$$

Since $b\ cost(2, j) = c(1, j)$ if $< 1, j > \in E$

So $b\ cost(2, 2) = c(1, 2) = 3$

$b\ cost(2, 3) = c(1, 3) = 2$

Now

$$b\ cost(3, 4) = \min\{b\ cost(2, 2) + c(2, 4), b\ cost(2, 3) + c(3, 4)\}$$
$$= \min\{3 + 4, 2 + 3\}$$
$$= \min\{7, 5\}$$
$$= 5$$
$$b\ cost(3, 5) = \min\{b\ cost(2, 2) + c(2, 5), b\ cost(2, 3) + c(3, 5)\}$$
$$= \min\{3 + 2, 2 + 1\}$$
$$= \min\{5, 3\}$$
$$= 3$$
$$b\ cost(4, 6) = \min\{b\ cost(3, 4) + c(4, 6), b\ cost(3, 5) + c(5, 6)\}$$
$$= \min\{5 + 6, 3 + 2\}$$
$$= \min\{11, 5\}$$
$$= 5$$

The minimum path cost from 1 to 6 is 5

and path will be $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$

Prob.9. Find all pair shortest path using Floyd-Warshall algorithm for the given graph.

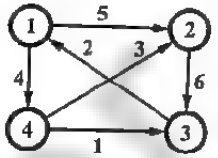


Fig. 3.11

(R.G.P.V., June 2011)

Sol. The sequence of matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the given graph is computed as follows –

$$D^{(0)} = \begin{pmatrix} 0 & 5 & \infty & 4 \\ \infty & 0 & 6 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & 3 & 1 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{pmatrix} NIL & 1 & NIL & 1 \\ NIL & NIL & 2 & NIL \\ 3 & NIL & NIL & NIL \\ NIL & 4 & 4 & NIL \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 5 & \infty & 4 \\ \infty & 0 & 6 & \infty \\ 2 & 7 & 0 & 6 \\ \infty & 3 & 1 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{pmatrix} NIL & 1 & NIL & 1 \\ NIL & NIL & 2 & NIL \\ 3 & 1 & NIL & 1 \\ NIL & 4 & 4 & NIL \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 5 & 11 & 4 \\ \infty & 0 & 6 & \infty \\ 2 & 7 & 0 & 6 \\ \infty & 3 & 1 & 0 \end{pmatrix} \quad \pi^{(2)} = \begin{pmatrix} NIL & 1 & 2 & 1 \\ NIL & NIL & 2 & NIL \\ 3 & 1 & NIL & 1 \\ NIL & 4 & 4 & NIL \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 5 & 11 & 4 \\ 8 & 0 & 6 & 12 \\ 2 & 7 & 0 & 6 \\ 3 & 3 & 1 & 0 \end{pmatrix} \quad \pi^{(3)} = \begin{pmatrix} NIL & 1 & 2 & 1 \\ 3 & NIL & 2 & 1 \\ 3 & 1 & NIL & 1 \\ 3 & 4 & 4 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 5 & 5 & 4 \\ 8 & 0 & 6 & 12 \\ 2 & 7 & 0 & 6 \\ 3 & 3 & 1 & 0 \end{pmatrix} \quad \pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 1 \\ 3 & NIL & 2 & 1 \\ 3 & 1 & NIL & 1 \\ 3 & 4 & 4 & NIL \end{pmatrix}$$

Prob.10. Use the Floyd-Warshall algorithm and find all pair shortest paths for the following adjacency weighted matrix.

$$\begin{bmatrix} 0 & 4 & \infty & 3 \\ \infty & 0 & 2 & 1 \\ 5 & 3 & 0 & \infty \\ 1 & \infty & 2 & 0 \end{bmatrix}$$

(R.G.P.V., June 2017)

Sol.

$$D^{(0)} = \begin{bmatrix} 0 & 4 & \infty & 3 \\ \infty & 0 & 2 & 1 \\ 5 & 3 & 0 & \infty \\ 1 & \infty & 2 & 0 \end{bmatrix} \quad \pi^{(0)} = \begin{bmatrix} NIL & 1 & NIL & 1 \\ NIL & NIL & 2 & 2 \\ 3 & 3 & NIL & NIL \\ 4 & NIL & 4 & NIL \end{bmatrix}$$

and for $D^{(1)}$ –

(i) Set row 1 as row 1 of $D^{(0)}$.

(ii) Set column 1 same as column 1 of $D^{(0)}$.

(iii) Set rows and columns of $D^{(1)}$ same as row and column of the which contains ∞ in current $D^{(1)}$. Here column 3 and row 2 will be same $D^{(0)}$.

(iv) Fill the other column according to the formula –

$$D_{ij}^{(k)} = \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right)$$

$$D^{(1)} = \begin{bmatrix} 0 & 4 & \infty & 3 \\ \infty & 0 & 2 & 1 \\ 5 & 3 & 0 & \mathbf{8} \\ 1 & \mathbf{5} & 2 & 0 \end{bmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & 2 & 2 \\ 3 & 3 & \text{NIL} & 1 \\ 4 & 1 & 4 & \text{NIL} \end{bmatrix}$$

Similarly for $D^{(2)}$ and so on upto $D^{(4)}$

$$D^{(2)} = \begin{bmatrix} 0 & 4 & \mathbf{6} & 3 \\ \infty & 0 & 2 & 1 \\ 5 & 3 & 0 & \mathbf{4} \\ 1 & 5 & 2 & 0 \end{bmatrix} \quad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & 2 & 2 \\ 3 & 3 & \text{NIL} & 1 \\ 4 & 1 & 4 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 & 3 \\ \mathbf{7} & 0 & 2 & 1 \\ 5 & 3 & 0 & 4 \\ 1 & 5 & 2 & 0 \end{bmatrix} \quad \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 2 & 1 \\ 3 & \text{NIL} & 2 & 2 \\ 3 & 3 & \text{NIL} & 2 \\ 4 & 1 & 4 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 4 & \mathbf{5} & 3 \\ \mathbf{2} & 0 & 2 & 1 \\ 5 & 3 & 0 & 4 \\ 1 & 5 & 2 & 0 \end{bmatrix} \quad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 1 \\ 4 & \text{NIL} & 2 & 2 \\ 3 & 3 & \text{NIL} & 2 \\ 4 & 1 & 4 & \text{NIL} \end{bmatrix}$$

$D^{(4)}$ is the required answer.

Prob.11. Using Floyd-Warshall algorithm solve the all pair shortest path problem for the graph. Where weight matrix is given below –

$$\begin{bmatrix} \boldsymbol{0} & \boldsymbol{\infty} & \boldsymbol{3} & \boldsymbol{\infty} \\ \boldsymbol{2} & \boldsymbol{0} & \boldsymbol{\infty} & \boldsymbol{\infty} \\ \boldsymbol{\infty} & \boldsymbol{7} & \boldsymbol{0} & \boldsymbol{1} \\ \boldsymbol{6} & \boldsymbol{\infty} & \boldsymbol{\infty} & \boldsymbol{0} \end{bmatrix}$$

(R.G.P.V., Dec. 2

Sol. The sequence matrix $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warsh algorithm for the given graph is computed below –

$$D^{(0)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & \text{NIL} & 1 & \text{NIL} \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & 3 & \text{NIL} & 3 \\ 4 & \text{NIL} & \text{NIL} & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & \text{NIL} & 1 & \text{NIL} \\ 2 & \text{NIL} & 1 & \text{NIL} \\ \text{NIL} & 3 & \text{NIL} & 3 \\ 4 & \text{NIL} & 1 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \quad \pi^{(2)} = \begin{bmatrix} \text{NIL} & \text{NIL} & 1 & \text{NIL} \\ 2 & \text{NIL} & 1 & \text{NIL} \\ 2 & 3 & \text{NIL} & 3 \\ 4 & \text{NIL} & 1 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \quad \pi^{(3)} = \begin{bmatrix} \text{NIL} & 3 & 1 & 3 \\ 2 & \text{NIL} & 1 & 3 \\ 2 & 3 & \text{NIL} & 3 \\ 4 & 3 & 1 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \quad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 3 & 1 & 3 \\ 2 & \text{NIL} & 1 & 3 \\ 4 & 3 & \text{NIL} & 3 \\ 4 & 3 & 1 & \text{NIL} \end{bmatrix}$$

Prob.12. What is dynamic programming ? Applying dynamic programming determine a longest common subsequence of <1, 0, 0, 1, 0, 1, 0, 1> and <0, 1, 0, 1, 1, 0, 1, 1, 0>. ***(R.G.P.V., Dec. 2011)***

Or

Determine on LCS of <1, 0, 0, 1, 0, 1, 0, 1 > and < 0, 1, 0, 1, 1, 0, 1, 1, 0>. How dynamic programming can be applied to LCS ? (R.G.P.V., Dec. 2012)

Sol. **Dynamic Programming** – Refe to Q.1.

Longest common subsequence of <1, 0, 0, 1, 0, 1, 0, 1 > and < 0, 1, 0, 1, 1, 0, 1, 1, 0, > is determined as follows –

	y_j	0	1	0	1	1	0	1	1	0
x_i	0	0	0	0	0	0	0	0	0	0
1	0	↑0	1	←1	1	1	←1	1	1	←1
0	0	1	↑1	2	←2	←2	2	←2	←2	2
0	0	1	↑1	2	↑2	↑2	3	←3	←3	3
1	0	↑1	2	↑2	3	3	↑3	4	4	←4
0	0	1	↑2	3	↑3	↑3	4	↑4	↑4	5
1	0	↑1	2	↑3	4	4	↑4	5	5	↑5
0	0	1	↑2	3	↑4	↑4	5	↑5	↑5	6
1	0	↑1	2	↑3	4	5	↑6	6	6	↑6

A longest common sequence could be <1, 0, 0, 1,1, 0> as illustrated below (there could be some other LCS', if we choose ← instead of ↑ when c[i – 1, j] = c[i, j – 1]);

X : <–, 1, 0, –, –, 0, 1, 0, 1, 0, 1 >

Y : <0, 1, 0, 1, 1, 0, 1, –, 1, 0, –>

So the LCS is 100110.

UNIT 4

BACKTRACKING CONCEPT AND ITS EXAMPLES LIKE 8 QUEEN'S PROBLEM, HAMILTONIAN CYCLE, GRAPH COLORING PROBLEM ETC.

Q.1. Explain backtracking. Write an algorithm to estimate the efficiency of backtracking. ***(R.G.P.V., Dec. 2012)***

Ans. Backtracking is the most general technique to solve many problems that deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints. The name backtrack was first coined by D.H. Lehmer in the 1950s.

In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, x_2, ..., x_n)$, where the x_i are chosen from some finite set S_i. Generally the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a ***criterion function*** $P(x_1, x_2,..., x_n)$. Sometimes it seeks all vectors that satisfy P.

Some of the examples that can be solved by backtracking are –

(i) Sorting the array of integers in a {1 : n}

(ii) 8-queens problem

(iii) 4-queens problem, or in generalized way n queens problem

(iv) Sum of subsets problem.

Algorithm – 4.1 Estimating the Efficiency of Backtracking

```
1.  Algorithm Estimate( )
2.  // This algorithm follows a random path
3.  // in a state space tree and produces an
4.  // estimate of the number of nodes in the tree.
5.  {
6.       i := 1; j := 1; k := 1;
7.  repeat
8.  {
```

```
9.        V_i:= {a[i] | a[i] ∈ V(a[1],a[2], ....., a[i − 1])
10.           and B_i (a [1], ....., a [i] is true};
11.       If (Size (V_i) = 0) then return j;
12.       k : = k * Size (V_i); j : = j + k;
13.       a [i] : = Choose (V_i); i: = i + 1;
14.       }   until (false);
15. }
```

Q.2. Write recursive backtracking algorithm.

Ans. Algorithm 4.2 presents a recursive formulation of the backtrack technique. Naturally describing backtracking in this way is essential becaus is a postorder traversal of a tree. This recursive version is initially invoked Backtrack (1) ;

Algorithm 4.2 Recursive Backtracking Algorithm

```
1.    Algorithm Backtrack (k)
2.    // This schema describes the backtracking process using
3.    // recursion. On entering, the first k-1 values.
4.    // x[1], x[2], ...., x[k − 1] of the solution vector.
5.    // x[1 : n] have been assigned. x[ ] and n are global.
6.    {
7.       for (each x [k] ∈ T (x [1], ....., x [k − 1]) do
8.          {
9.             if (B_k (x[1], x[2], ....., x[k] ≠ 0) then
10.               {
11.                  if (x [1], x[2], ....., x [k] is a path to an answer no
12.                     then write (x [1 : k]);
13.                  if (k < n) then Backtrack (k +1);
14.               }
15.         }
16.   }
```

The solution vector $(x_1, x_2, \ldots, x_k)$ is treated as a global array x [1 : All the possible elements for the k^{th} position of the tuple that satisfy B_k generated, one by one and then they are adjoined to the current vector (x_1, x_{k-1}). Each time x_k is attached, a check is made to find whether a solution been found. Then the algorithm 4.1 is recursively invoked. When the for loop line 7 is completed, no more values for x_k exist and the current copy of backtr ends. The last unresolved calls now resumes, namely, the one that continues see the remaining elements assuming only k–2 values have been set.

Here point to be noted is that this algorithm causes ***all*** solutions to printed and assumes that tuples of various sizes may make up a solution.

Q.3. Write non-recursive backtracking algorithm.

Ans. An iterative or non-recursive version of algorithm 4.2 appear in algorithm 4.3.

Algorithm 4.3 General Iterative or Non-recursive Backtracking Method

```
1.    Algorithm Backtrack (k)
2.    // This schema describes the backtracking process.
3.    // All solutions are generated in x [1 : n] and printed
4.    // as soon as they are determined.
5.    {
6.       k : = 1 ;
7.       while (k ≠ 0) do
8.       {
9.          if (there remains an untried x[k] ∈ T (x [1], x[2], ....,
10.            x [k − 1]) and B_k (x [1], ....., x[k]) is true) then
11.         {
12.         if (x [1], ....., x[k] is a path to an answer node)
13.            then write (x [1 : k]);
14.            k : = k + 1;   // consider the next set
15.         }
16.         else k : = k − 1; // backtrack to the previous set
17.      }
18.   }
```

Here point to be noted is that T( ) will yield the set of all possible values that can be placed as the first component x_1 of the solution vector. The component x_1 will take values for which the bounding function $B_1(x_1)$ is true. Also notice that how the elements are generated in a depth first manner. The variable k is continually incremented and a solution vector is grown until either a solution is found or no untried value of x_k remains. The algorithm 4.3 must resume the generation of possible elements for the kth position that have not yet been tried when k is decremented. Hence one must develop a procedure that generates these values in some order.

Q.4. What is backtracking ? Explain recursive and non-recursive backtracking. ***(R.G.P.V., Dec. 2011)***

Or

Explain the concept of backtracking. ***(R.G.P.V., June 2008, 2014)***

Or

Explain backtracking technique for designing an algorithm. ***(R.G.P.V., Dec. 2007, June 2013)***

Or

Explain backtracking in detail. Also write algorithm for recursive backtracking algorithm. ***(R.G.P.V., June 2015)***

Ans. Refer to Q.1, Q.2 and Q.3.

Q.5. Explain state space tree. ***(R.G.P.V., June 201***

Ans. State space tree is useful to develop some terminology regard... tree organizations of solution spaces. Each node in this tree defines a probl... state, All paths from the root to other nodes define the state space of ... problem. Solution states are those problem states S for which the path fro... the root to S defines a tuple in the solution space. Answer states are tho... solution states S for which the path from the root to S defines a tuple that i... member of the set of solutions of the problem. The tree organization of t... solution space is referred to as the state space tree.

Q.6. Explain 4-queens problem.

Ans. Similar to n-queens or 8-queens problem, in 4-queens problem w... have 4-queens to be placed on a 4 × 4 chessboard, satisfying the constrai... that no two queen should be in same row, same column, or in same diagon...

Now according to external constraints the solution space consists of ... 4-tuples i.e., $S_i = \{1, 2, 3, 4\}$ and $1 < i < 4$, whereas the internal constra... have 4! solutions i.e., permuation of 4.

Fig. 4.1 shows the tree organization of the 4-queens solution space.

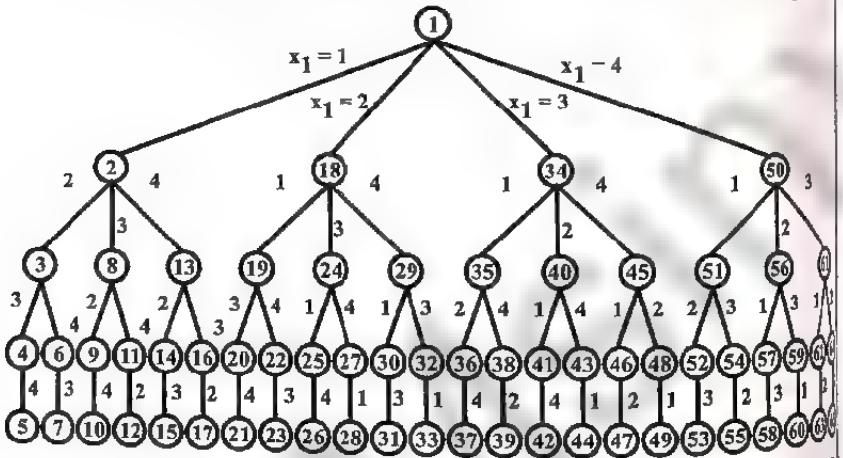


Fig. 4.1 Tree Organization of the 4-queens Solution Space (Nodes are Numbered in Depth First Search)

The tree shown in fig. 4.1 is called the permutation tree. The edges ... labeled by possible values of x_i. Edges from level 1 to level 2 specify the va... for x_1. Hence the leftmost subtree contains all solutions with $x_1 = 1$; its leftmo... subtree contains all solutions with $x_1 = 1$ and $x_2 = 2$, and so on. Edges fro... level i to level i + 1 are labeled (represented) with the values of x_i. Thus t... solution space is defined by all paths from the root node to a leaf node. The... are 4! = 24 leaf nodes in the tree of above figure.

Q.7. Explain and solve 4-queen's problem using backtracking. ***(R.G.P.V., June 2010, Dec. 2017)***

Ans. Refer to Q.6.

Backtracking method can be applied on 4-queens problem to solve it. In this technique, as a bounding function, we use the obvious criterion that if $(x_1, x_2, ..., x_i)$ is the path to the current E-node, then all the children nodes with parent-child labelings x_{i+1} are such that –

$(x_1, x_2, ..., x_{i+1})$ represents a chessboard configuration in which no two queens are attacking.

We start with the root node as the only live node. This time this node becomes the E-node and the path is ( ). We generate the next child. Suppose we are generating the child in ascending order. Thus, node number 2 of fig. 4.1 is generated and the path is now 1. See fig. 4.2.

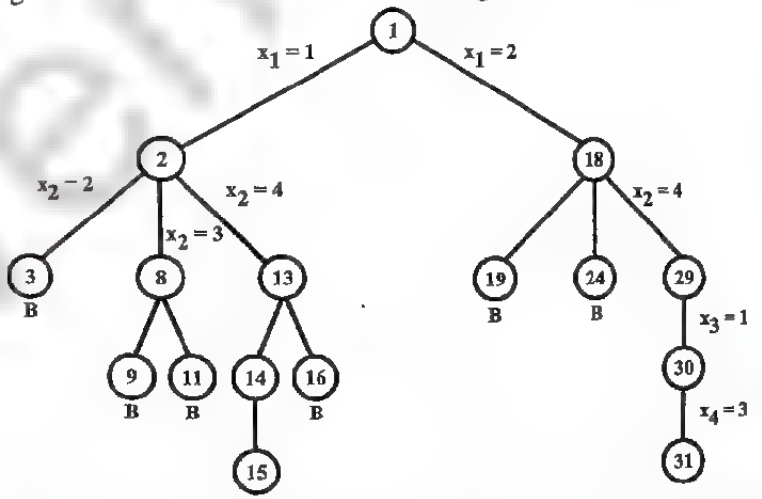


Fig. 4.2 Portion of the Tree of Fig. 4.1 that is Generated During Backtracking

This means that the queen 1 is placed in first row and in first column as shown in fig. 4.3(a).

Now, node 2 becomes the next E node or line node. Further, try next ascending node that node is 3 having $x_2 = 2$. i.e., queen 2 placed in 2nd column but if we do this then queen 1 and queen 2 become in same diagonal, so node 3 becomes dead here, and we backtrack to node 2 and try next possible node, i.e., node 8.

Here, $x_2 = 3$, i.e., queen 2 is placed in 3rd column as shown in fig. 4.3(b). This satisfy all the constraints. So now, node 8 becomes the next live node.

After this we try node 9 having $x_3 = 2$, i.e., queen 3 placed in 2nd column, but by this, queen 2 and queen 3 are in same diagonal. So, this node becomes dead.

We try for next possible node 11, with $x_3 = 4$. But in this case also queen 2 and queen 3 are in same diagonal resulting node 11 as a dead node shown in fig. 4.2 indicated by symbol B.

We have tried all possible positions for queen 3, i.e., column 1, 2, 3, but any of positions is not satisfying all the constraints, shown in fig. 4.3 (c). So, we backtrack to previous live node, i.e., node 2 and try another possible node, i.e., node 13.

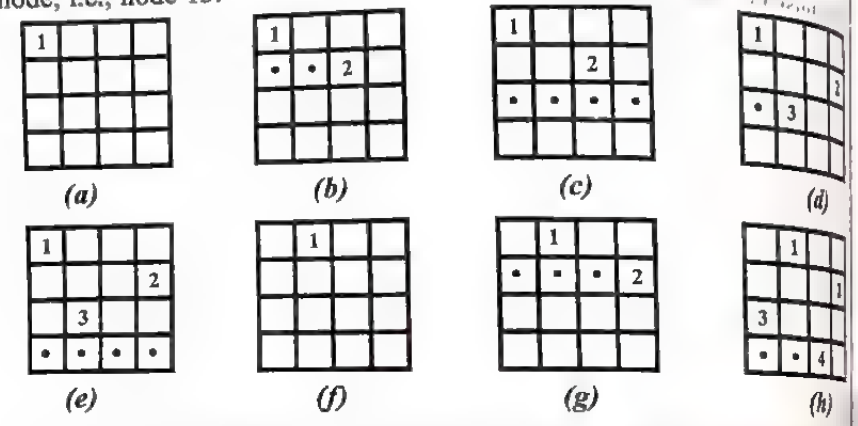


Fig. 4.3 Solution of 4-queens Problem using Backtracking

Now, node 13 becomes the new live node with $x_2 = 4$, i.e., queen 2 is placed in 4th column as shown in fig. 4.3 (d).

After this we try next node, i.e., node 14. It becomes next live node with $x_3 = 2$, i.e., queen 3 is placed in 2nd column as shown in fig. 4.3 (d). Further we go ahead with node 15 as new live node with $x_4 = 3$. But this makes queen 3 and queen 4 placed in same diagonal resulting in a dead node 15, and we backtrack to node 14, shown in fig. 4.3 (e), and then backtrack to node 13 and try another possible node 16, with $x_3 = 3$ but this results in queen 2 and queen 3 in same diagonal. So, this node also becomes dead.

We further backtrack to node 2, but no other node is left to try, so node 2 is also killed and we backtrack to node 1 and try another subtree having $x_1 = 2$, i.e., queen 1 placed in 2nd column shown in fig. 4.3 (f).

Again with similar reason, nodes 19 and 24 are killed and we try node 7 with $x_2 = 4$, i.e., queen 2 placed in 4th column, shown in fig. 4.3 (g). Then we try node 30 as next live node with $x_3 = 1$ and finally proceed to node 31 with $x_4 = 3$, i.e., queen 4 placed in 3rd column, as shown in fig. 4.3 (h).

Here, all the constraints are satisfied, so we have desired result {2, 4, 1, 3}, for 4-queens problem.

Q.8. What is backtracking ? Find a solution to the 4-queen problem using backtracking strategy. ***(R.G.P.V., Dec. 2009)***

Ans. Refer to Q.1 and Q.7.

Q.9. What is 8-queens problem ?

Ans. 8-queens problem is to place eight queens on an 8 × 8 chessboard so that no two "attack", i.e., so that no two of them are on the same row, column, or diagonal. Let us number the rows and columns of the chessboard 1 to 8 as shown in fig. 4.4. The queens can also be numbered 1 to 8. We can without loss of generality assume queen i is to be placed on row i, since each queen must be on a different row. Therefore, all solutions to the 8-queens problem can be represented as 8-tuples $(x_1, \ldots\ldots, x_8)$, where x_i is the column on which queen i is placed. The explicit constraints for this formulation are $s_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \le i \le 8$. Therefore, the solution space is made of 8^8, 8 tuples. The implicit constraints for this problem are that no two x_i's can be the same and no two queens can be on the same diagonal. The first of these two constraints specifies that all solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8). This realization decreases the size of the solution space from 8^8 tuples to 8! tuples. Expressed as an 8-tuple, the solution in fig. 4.4 is (4, 6, 8, 2, 7, 1, 3, 5).

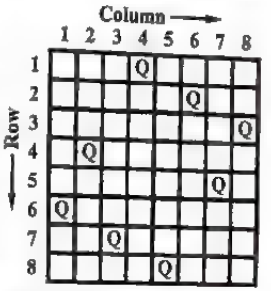


Fig. 4.4 One Solution to the 8-queens Problem

Q.10. Explain backtracking. How backtracking algorithm can be used to solve 8 queens problem ? ***(R.G.P.V., Dec. 2009, 2011, 2013)***

Or

Explain eight queen's problem and apply backtracking to solve this problem. ***(R.G.P.V., June 2016, Nov. 2018)***

Or

What is backtracking ? Explain 8 queen's problem and how can we solve it using backtracking. ***(R.G.P.V., May 2019)***

Ans. **Backtracking** – Refer to Q.1.

In 8-queens problem we are supposed to place 8 queens on a 8 × 8 chessboard so that no one attacks each other. This problem can be solved by backtracking. Here, we are discussing a different generalized way to solve such problems.

Any chessboard square is numbered as 1....n, in two dimension, i.e., a two dimensional matrix a[1:n, 1:n] having n rows and n columns, as shown in fig. 4.5.

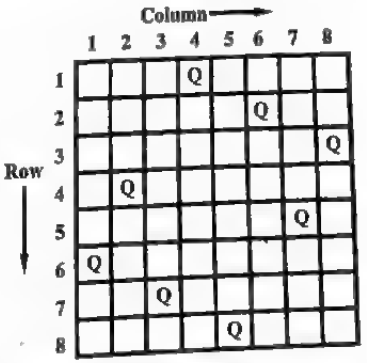


Fig. 4.5 Solution of 8-queens Problem

One general property of such matrix is that every element on the same diagonal that runs from the upper left to the lower right has the same row column value. For example, consider the queen at a[3,2] from fig. 4.5. The diagonal squares of this queen are a[3,2], a[4,3], a[5,4], a[6,5], and so on. One property that relates them is that difference of rows and columns is constant value.

i.e. $3 - 2 = 1$

$4 - 3 = 1$

$5 - 4 = 1$

$6 - 5 = 1$, and so on.

Similarly, for the elements that run from the upper right to lower left have the same row + column values. For example, in fig. 4.5, diagonal elements are a[1,8], a[2,7], a[3,6], a[4,5], and so on.

i.e., $1 + 8 = 9$

$2 + 7 = 9$

$3 + 6 = 9$

and $4 + 5 = 9$.

So, we can use above property to determine the result of our problem.

Say, any queen is diagonally placed to another queen if and only if –

$$i - j = k - l$$

or

$$i + j = k + l$$

where two queens are placed at positions (i, j) and (k, *l*). The first equation implies –

$$j - l = i - k$$

and second equation implies

$$j - l = k - i$$

So, two queens are in same diagonal if

$$|j - l| = |i - k|.$$

Algorithm for solving n-queens problem can be given by using the property that elements are on same diagonal if

$$|j - l| = |i - k|.$$

Algorithm 4.4 Gives Answer Whether a New Queen can be Placed in k^{th} Row and i^{th} Column

```
1.  Algorithm Place(k, i)
2.  // Returns true if a queen can be placed in k^th row and
3.  // i^th column. Otherwise it returns false. x[] is a
4.  // global array whose first (k – 1) values have been set.
```



```
5.  // Abs(r) returns the absolute value of r.
6.  {
7.      for j := 1 to k – 1 do
8.          if ((x[j] = i) // Two in the same column
9.                  or (Abs (x[j] – i) = Abs(j – k)))
10.                     // or in the same diagonal
11.                 then return false;
12.     return true;
13. }
```

Here, algorithm 4.4 shows whether a queen can be placed in k^{th} row and i^{th} column. If it can, then algorithm returns true otherwise returns false.

In this, we test both whether it is distinct from all previous values x[1],.....,x[k–1] and whether there is no other queen on the same diagonal. Its computing time is O(k – 1).

Algorithm 4.5 All Solutions to the n-queens Problem

```
1.  Algorithm NQueens(k, n)
2.  // Using backtracking, this procedure prints all
3.  // possible placements of n queens on an n × n
4.  // chessboard so that they are nonattacking.
5.  {
6.      for i := 1 to n do
7.      {
8.          if Place(k, i) then
9.          {
10.                 x[k] := i;
11.                 if (k = n) then write (x[1 : n];
12.                 else NQueens(k + 1, n);
13.         }
14.     }
15. }
```

And in algorithm 4.5, we print all possible placements of n queens on an n × n chessboard. So by using backtracking, no two of them are attacking each other.

By applying above algorithm some of the possible walks for 8-queens problem are shown in fig. 4.6.

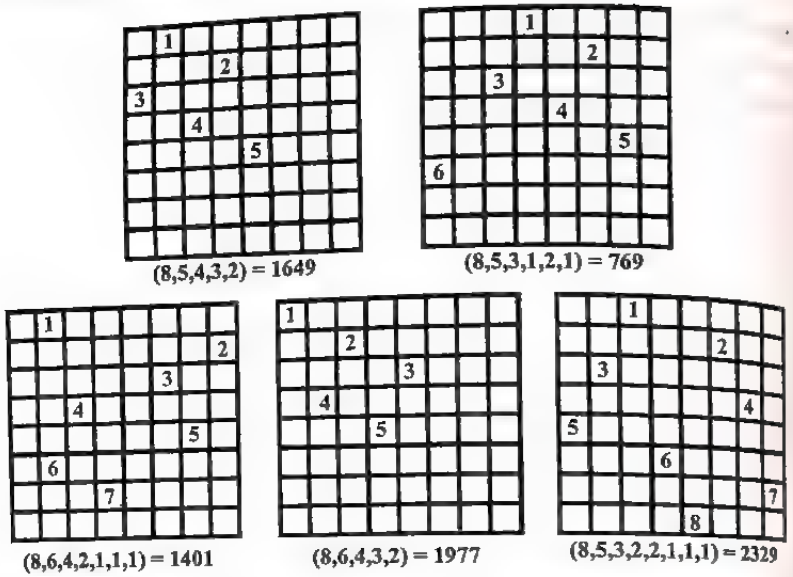


Fig. 4.6 Five Walks through the 8-queens Problem Plus Estimates of the Tree Size

Q.11. Explain n-queens problem. ***(R.G.P.V., June 2019)***

Ans. The n-queens problem is a generalized problem of 8-queens or 4-queens problem. Here, we can think that there are n queens to be placed on a n × n chessboard. That means, we have a chessboard having n rows and n columns, and n queens are to be placed on this n × n chessboard such that no two queens are in same row or in same column or in same diagonal. That is, no two queens 'attack' each other.

Here, we suppose that queen i is to be placed in row i. Say, 1 queen will be placed in first row only, but can have any column from 1,2,..., n so that satisfy the explicit and implicit constraints (See fig. 4.7).

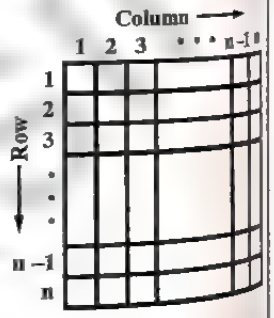


Fig. 4.7 An n × n Chessboard

All solutions to the n-queens problem can therefore be represented as n-tuples (x_1, x_2, ..., x_n) where x_i is the column on which queen i is placed.

The explicit constraints using this formulation are $S_i = \{1,2,3,.....,n-1,n\}$, where $1 \le i \le n$. Therefore, the solution space consists of n^n n-tuples. Now, considering the implicit constraints, that no two x_i's can be the same i.e., two queens cannot be in same row, same column, or in same diagonal. So, each x_i should be different. So, by above constraint, our solution space can be reduced

as all solutions are permutations of the n-tuple (1, 2, 3, ..., n – 1, n). So by this constraint, our solution space reduces from n^n tuples to n! tuples.

Based on above constraints using backtracking technique, we can solve our problem.

Q.12. Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1). ***(R.G.P.V., Dec. 2015)***

Ans. The given sequence represents column position and each queen is placed row by row on given column. That is place 1st queen at (1, 6), 2nd queen at (2, 4) and so on.

Queen Positions								*Action*
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
6	4	7	1					Start
6	4	7	1	2				Not Feasible ∵ 5 – 4 = 2 – 1
6	4	7	1	3				
6	4	7	1	3	2			Not Feasible ∵ 6 – 5 = 3 – 2
6	4	7	1	3	5			
6	4	7	1	3	5	2		
6	4	7	1	3	5	2	8	List Ends : Feasible Sequence

Hence solution to 8-queen's problem is (6, 4, 7, 1, 3, 5, 2, 8).

Q.13. Give the definition of Hamiltonian cycle.(R.G.P.V., June 2016)

Ans. Let G = (V, E) be a connected graph with n vertices. A ***Hamiltonian cycle*** is a round-trip path along n edges of G that visits every vertex once and returns to its initial or starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, ...,v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i < n$, and the v_i are different except for v_1 and v_{n+1}, which are equal. Hamiltonian cycle was suggested by Sir William Hamilton.

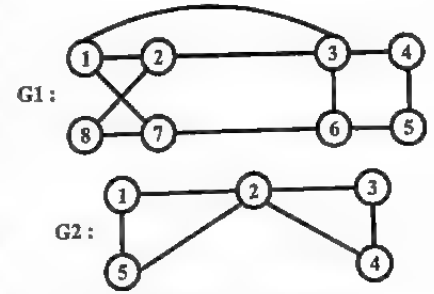


Fig. 4.8 Two Graphs, One Containing a Hamiltonian Cycle

Fig. 4.8 shows a graph G1 which contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G2 does not contain any Hamiltonian cycle. There is no easy way to find whether a given graph contains a Hamiltonian cycle. We have backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

Q.14. Find a Hamiltonian circuit using backtracking method.

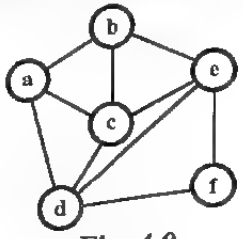
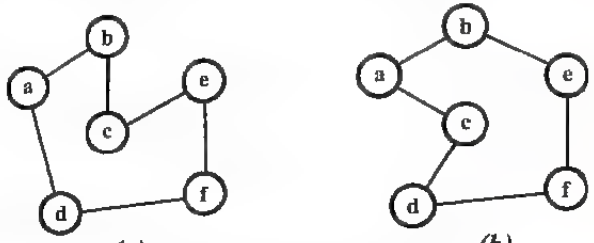


Fig. 4.9 *(R.G.P.V., Dec. 2010)*

Ans. In a given graph more than one Hamiltonian circuit exists, which are as follows –



(a) *(b)*

Fig. 4.10 Hamiltonian Cycles

Q.15. Explain Hamiltonian cycle with suitable example. *(R.G.P.V., Dec. 2012)*

Or

Write short note on Hamiltonian cycle. (R.G.P.V., June 2008, May 2019)

Or

Define Hamiltonian cycle with example. *(R.G.P.V., Dec. 2016)*

Ans. Refer to Q.13 and Q.14.

Q.16. What is Hamiltonian cycle ? Write an algorithm to find all Hamiltonian cycles in a graph. *(R.G.P.V., June 2007, Dec. 2008, June 2010, 2011, 2013, Dec. 2013)*

Or

What is Hamiltonian cycle ? Explain how it can be solved using backtracking algorithm. *(R.G.P.V., June 2014, Dec. 2014, 2015)*

Ans. **Hamiltonian Cycle** – Refer to Q.13.

Algorithm – The backtracking solution vector $(x_1, ...,x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. Now we have to find how to compute the set of possible vertices for x_k if $x_1, ..., x_{k-1}$ have already been chosen. If $k = 1$ the x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, ...,x_{k-1}$ and v is connected by an edge to x_{k-1}. Only the vertex x_n be the one remaining vertex and it must be connected to both x_{n-1} and x_1. Algorithm 4.6 NextValue (k) determines a possible next vertex of the proposed cycle.

Algorithm 4.6 Generating a Next Vertex

```
1.  Algorithm NextValue (k)
2.  // x [1 : k – 1] is a path of k – 1 distinct vertices. If
3.  // x [k] = 0, then no vertex has as yet been assigned to x [k]. After
4.  // execution, x [k] is assigned to the next highest numbered vertex
5.  // which does not already appear in x [1 : k – 1] and is connected by
6.  // an edge to x [k – 1]. Otherwise x [k] = 0. If k = n, then
7.  // in addition x [k] is connected to x [1].
8.  {
9.      repeat
10.     {
11.         x[k] : = (x [k] + 1) mod (n+1); //Next vertex.
12.         if (x [k] = 0) then return;
13.         if (G[x [k – 1], x [k]] ≠ 0) then
14.         {// Is there an edge ?
15.         for j : = 1 to k – 1 do if (x [j] = x[k]) then break;
16.                     // Check for distinctness
17.         if ( j = k) then // If true, then the vertex is distinct.
18.                     if ((k < n) or ((k = n) and G [x [n], x [1]] ≠ 0))
19.                       then return ;
20.         }
21.     } until (false) ;
22. }
```

Algorithm 4.7 is used to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G [1 : n, 1 : n], then setting x [2 : n] to zero and x[1] to 1, and then executing Hamiltonian (2).

Algorithm 4.7 Finding All Hamiltonian Cycles

```
1.  Algorithm Hamiltonian (k)
2.  // This algorithm uses the recursive formulation of
3.  // backtracking to find all the Hamiltonian cycles
4.  // of a graph. The graph is stored as an adjacency
5.  // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6.  {
7.      repeat
8.      {  // Generate values for x [k].
9.         NextValue (k);              //Assign a legal next value to x[k].
10.        if (x [k] = 0) then return;
11.        if (k = n) then write (x [1 : n]);
12.        else Hamiltonian (k + 1);
13.     } until (False) ;
14. }
```

Traveling salesman problem is a problem of finding tour with minimum cost. This tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are similar, Hamiltonian will determine a minimum-cost tour if a tour exists. If the common edge cost is C, the cost of a tour is cn as there are n edges in a Hamiltonian cycle.

Hamiltonian Cycle's Complexity – We can define hamiltonian-cycle as a formal language as –

$$\text{HAM-CYCLE} = \{\langle G\rangle : G \text{ is a hamiltonian graph}\}$$

If we use the reasonable encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G\rangle|$ is the length of the encoding of G. There are m! possible permutations of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, which is not $\theta(n^k)$ for any constant k.

Q.17. Apply and explain the backtracking method to solve the following-
(i) Hamiltonian circuit problem (ii) Subset-sort problem.
(R.G.P.V., May 2018)

Ans. (i) Hamiltonian Circuit Problem – Refer to Q.16.

(ii) Subset-sort Problem – The subset-sort problem or the subset sum problem is used to find the subset of elements which are selected from a given set and whose sum adds up to a given number K. Let us consider a set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented). Backtracking algorithm for subset sum is used to make a systematic consideration of the elements to be selected.

Let us assume a given set of 4 elements, such as W[1], W[2], W[3] and W[4]. The following tree diagram depicts approach of generating variable sized tuple and is used to design backtracking algorithm.

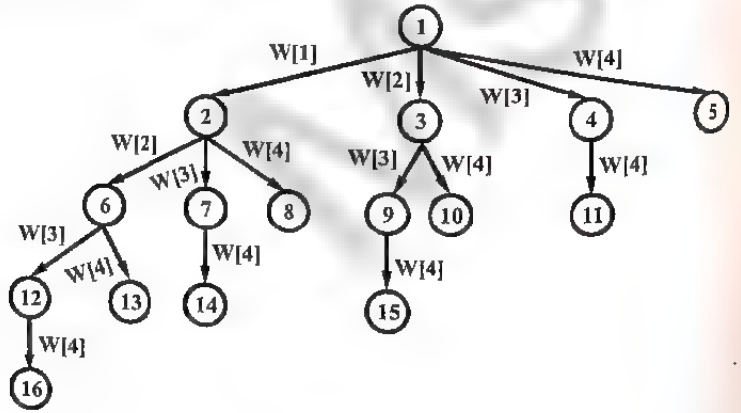


Fig. 4.11

The tree shows that the node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple element to be considered. For example, at level 1, tuple_vector [1] can take any value of four branches generated. At level 2 of left most node, tuple_vector [2] can take any value of three branches generated, and so on.

For example, the left most child of root generates all those subsets that include W[1]. Similarly the second child of root generates all those subsets that includes W[2] and excludes W[1].

Along depth of tree the elements are added and the sum of the elements is satisfying explicit constraints and further continue to generate child nodes. Whenever the constraints are not met, stop the further generation of subtrees of that node and backtrack to previous node to explore the nodes not yet explored. In many ways it saves considerable amount of processing time.

Q.18. Represent the map in the form of planar graph. Apply graph-colouring algorithm.

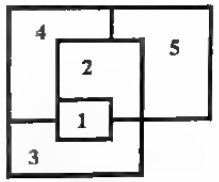


Fig. 4.12

(R.G.P.V., May/June 2006)

Or

Explain graph coloring problem with example. ***(R.G.P.V., Dec. 2010)***

Or

Explain graph coloring problem. ***(R.G.P.V., June 2011, 2015)***

Or

Write a short note on graph coloring. ***(R.G.P.V., May 2018)***

Or

Write short note on graph coloring problem. ***(R.G.P.V., May 2019)***

Ans. Let G be a graph and m be a positive integer. The graph coloring problem is to discover whether the nodes of G can be covered in such a way that no two adjacent nodes have the same color yet only m colors are used. This is also known as ***M-colorability decision*** problem. Here notice that if d is the degree of the given graph, then it can be colored with d+1 colors.

The ***M-colorability optimization*** problem deals with the smallest integer m for which the graph G can be colored. The integer is known as ***chromatic***

number of the graph. For example, the graph of fig. 4.12 can be colored with three colors 1, 2 and 3. The color of each node is indicated next to it. Here three colors are needed to color this graph and hence this graph's chromatic number is 3.

A graph is known to be ***planar*** iff it can be drawn in a plane in such a way that no two edges cross each other. A special case is the 4-color problem for planar graphs. The problem is to color the region in a map in such a way that no two adjacent regions have the same color. Yet only four colors are needed. This is a problem for which graphs are very useful because a map can be easily transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, they are joined by an edge. Fig. 4.13 shows a map with five regions and its corresponding graph. This map needs 4 colors. For many years, it is believed that five colors are needed to color any map but later mathematicians with the help of computer proofs that four colors are sufficient.

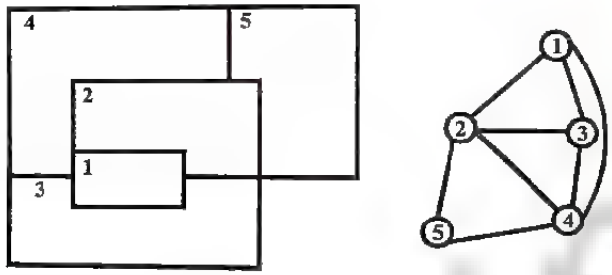


Fig. 4.13 A Map and Its Planar Graph Representation

Graph coloring problem can also be solved using state space tree, where, by applying backtracking method required results can be obtained.

Here, we have a state space tree with degree m and height n + 1, as shown in fig. 4.14.

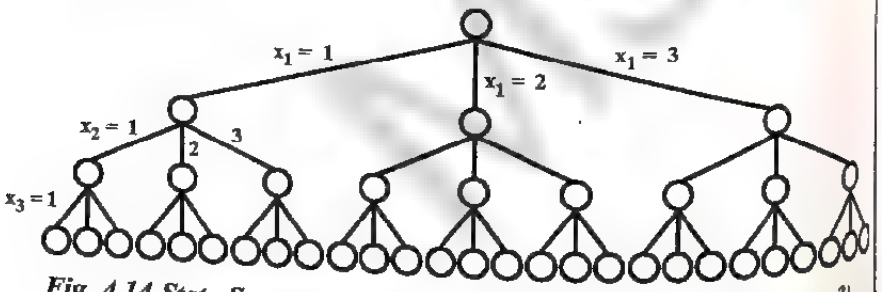


Fig. 4.14 State Space Tree for m Coloring (when n = 3 and m = 3)

Colors are indicated by integers (1,2,3) and solutions are given by fixed size n-tuples $(x_1, x_2, \ldots\ldots, x_n)$, where x_i is the color of node i, i.e., $x_1 = 1$

represents that node 1 is assigned color whose indices is 1. Similarly, $x_2 = 3$ denotes that node 2 is assigned with a color represented by indices 3.

In state space tree, each node at level i has m children corresponding to the m possible assignments to $x_i, 1 \le i \le n$. Nodes at level n + 1 are leaf nodes.

Q.19. Design a backtracking for graph coloring problem.
(R.G.P.V., Dec. 2009, 2017)

Or

Write a pseudo algorithm for graph coloring problem.
(R.G.P.V., June 2014, Dec. 2015)

Ans. For solving the graph coloring problem, we suppose that graph is represented by its adjacency matrix G[1:n, 1:n], where G[i, j] = 1 if (i, j) is an edge of G, and G[i, j] = 0 otherwise.

The colors are represented by the integers 1, 2, ..., m and the solutions are given by the n-tuple $(x_1, x_2, \ldots\ldots, x_n)$, where x_i is the color of node i. Algorithm for solving this problem is given in algorithm 4.8.

Algorithm 4.8 Finding all m-colorings of a Graph

```
1.  Algorithm mColoring(k)
2.  // This algorithm was formed using the recursive backtracking
3.  // schema. The graph is represented by its boolean adjacency
4.  // matrix G[1 : n,1 : n]. All assignments of 1,2,..., m to the
5.  // vertices of the graph such that adjacent vertices are
6.  // assigned distinct integers are printed. k is the index
7.  // of the next vertex to color.
8.  {
9.      repeat
10.     {// Generate all legal assignments for x[k].
11.         NextValue(k); // Assign to x[k] a legal color.
12.         if (x[k] = 0) then return; // No new color possible
13.         if (k = n) then // At most m colors have been
14.                                 // used to color the n vertices.
15.                 write (x[1 : n]);
16.         else mColoring (k + 1);
17.     } until (false);
18. }
```

Algorithm 4.8 uses recursive backtracking schema. In this algorithm, colors to be assigned are to be determined from the range (0, m), i.e., m colors are available. Now, the assignment of next color to the k^{th} node is represented by x[k]. This color is determined by algorithm 4.9, i.e., next value (k). It assumes that colors are assigned to k – 1 nodes in the range (1, m) such that adjacency matrix have different integers.

Algorithm 4.9 Generating a Next Color

```
1.  Algorithm NextValue(k)
2.  // x[1],...,x[k – 1] have been assigned integer values in
3.  // the range [1, m] such that adjacent vertices have distinct
4.  // integers. A value for x[k] is determined in the range
5.  // [0, m]. x[k] is assigned the next highest numbered color
6.  // while maintaining distinctness from the adjacent vertices
7.  // of vertex k. If no such color exists, then x[k] is 0.
8.  {
9.      repeat
10.     {
11.         x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12.         if (x[k] = 0) then return; // All colors have been used.
13.         for j := 1 to n do
14.         {           // Check if this color is
15.                     // distinct from adjacent colors.
16.                     if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17.                     // If (k, j) is an edge and if adj.
18.                     // vertices have the same color.
19.                     then break;
20.         }
21.         if (j = n + 1) then return; // New color found
22.     } until (false); // Otherwise try to find another color.
23. }
```

Now, value x[k], i.e., the color to be assigned to node k, is determined from the range (0, m), such that no two adjacent nodes have same values. If no such color exists then x[0] is zero, otherwise have appropriate value.

After this algorithm we assign the value of x[k] to k^{th} node, and checks whether k = n, i.e., whether all the nodes have been assigned proper color, if not then continue, otherwise stop the processing.

Total time required by above algorithm is $O(nm^n)$ including both algorithm 4.8 and algorithm 4.9.

Q.20. What is graph colouring problem ? Give an algorithm for colouring of a graph. ***(R.G.P.V., Dec. 2012)***

Or

What is graph coloring problem ? Give algorithm to solve this problem. ***(R.G.P.V., Dec. 2011, June 2013)***

Ans. Refer to Q.18 and Q.19.

Q.21. Explain graph coloring problem with their complexity. ***(R.G.P.V., Dec. 2016)***

Ans. **Graph Coloring Problem** – Refer to Q.18.

Complexity – Refer to Q.19.

Q.22. Explain how graph coloring problem can be solved using state space tree, with the help of an example.

Ans. Suppose, we have a graph G = (V, E), shown in fig. 4.15.

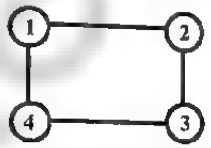


Fig. 4.15 A Connected Graph

The graph is a connected graph with 4 nodes. We have to assign each node a color such that no two adjacent nodes have the same color. This is possible by using atmost 3 colors. For this, a tree can be generated as shown in fig. 4.16, showing all possible ways in the problem to be solved.

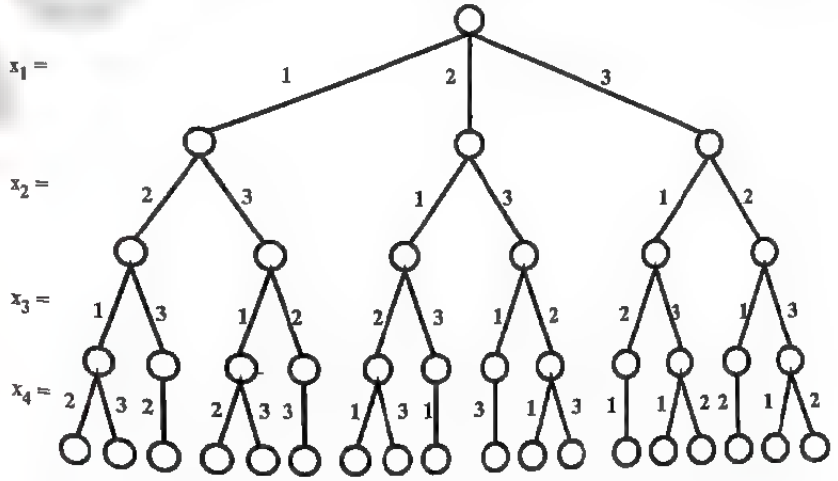


Fig. 4.16 All Possible 3 Colorings Tree having 4 Nodes

Each path to a leaf represents a color using atmost three colors. Here, only 12 solutions exist with exactly three colors. For example in rightmost subtree $x_1 = 3$, node 1 is assigned a color index 3 and node 2 is assigned a color index 2, i.e., $x_2 = 2$ and finally $x_3 = 3$ then only two possible colors are left, i.e., 1 and 2 that can be assigned to node 4.

So, $x_4 = 1$ or $x_4 = 2$.

Similarly, all other subtrees show the possible solutions.

NUMERICAL PROBLEMS

Prob.1. Colour the following graph using a vertex colouring algorithm. What is the minimum number of colours required ?

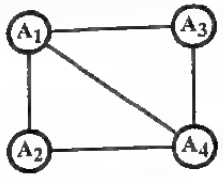


Fig. 4.17 *(R.G.P.V., June 2017)*

Sol. Solution of graph using vertex coloring algorithm which uses backtracking.

First create the adjacency matrix of graph –

$$M = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Initially colour of each vertices is 0.

Now, set the colour of $A_1 = 1$ and check weather its adjacent vertices have same colour to A or not.

Since $M[1, 2] = 1$

i.e., there is a path between A_1 and A_2

Now check the colour of A_2

we have $A_2 - 0 \neq A_1$ not same colour

Similarly check colour of A_3 and A_4

Colour of A_1 is not same as its adjacent vertices

So now go to next vertex.

Set colour of $A_2 = 1$

According of adjacency matrix

A_2 have A_1 and A_4 as its adjacent vertex

And colour of A_1 = colour of $A_2 = 1$

So, we can't go forward, we have to backtrack

Set colour of $A_2 = 2$ and compare its colour with colour of its adjacent vertex using adjacency matrix.

Result – no colour same

Now, set the A_3 vertex as colour 1 (i.e., $A_3 = 1$)

According to adjacency matrix A_3 have A_1 & A_4 as its adjacent vertices

$A_3 = A_1 = 1$

and

So, backtrack

Set $A_3 = 2$

No, adjacent vertex of A_3 have colour 2

Now set $A_4 = 1$

Adjacent vertex of $A_4 = A_1, A_2$ and A_3

and $A_1 = A_4 = 1$ have same colour

So backtrack

Set $A_4 = 2$

Adjacent vertex of $A_4 = A_1, A_2$ and A_3

and $A_2 = A_3 = A_4 = 2$

So backtrack

Set $A_4 = 3$

Adjacent vertex of $A_4 = A_1, A_2$ and A_3

No adjacent vertex of A_4 have colour 3

There is no vertex left so break.

Hence solution Vertex A_1 – colour 1
Vertex A_2 = colour 2
Vertex A_3 = colour 2
Vertex A_4 = colour 3

Minimum colour required = 3

Prob.2. Colour the following graph using a vertex colouring algorithm. What is the minimum colours required ?

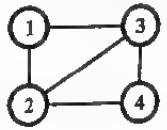


Fig. 4.18 *(R.G.P.V., Nov. 2018)*

Sol. Similar as Prob.1.

INTRODUCTION TO BRANCH AND BOUND METHOD, EXAMPLES OF BRANCH AND BOUND METHOD LIKE TRAVELLING SALESMAN PROBLEM ETC.

Q.23. Define branch and bound method. *(R.G.P.V., June 2016)*

Or

Explain the use of bounding function. *(R.G.P.V., Dec. 2014)*

Or

Write short note on branch and bound. *(R.G.P.V., Dec. 2017)*

Ans. The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node

can become the E-node. Two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalize to branch-and-bound strategies. In branch-and-bound terminology, a BFS-like state space search will be called FIFO search as the list of live nodes is a first-in-first-out list. A D-search-like state space search will be called LIFO search as the list of live nodes is a last-in-first-out list. Bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

Q.24. Explain how branch-and-bound method can be used to solve any problem. Give example.

Or

Explain the branch and bound search. Give an example.

(R.G.P.V., Dec. 2007)

Ans. We try to solve 4-queens problem using branch-and-bound technique. The problem is to place 4 queens on a 4 × 4 chessboard such that no two queens lie in same row, or in same column, or in same diagonal i.e., no two queens can attack to each other.

This constraint will act as bounding function in our branch-and-bound technique. We will use it to kill the subtrees that does not contain an answer node. State space tree of 4 queens problem is shown in fig. 4.19.

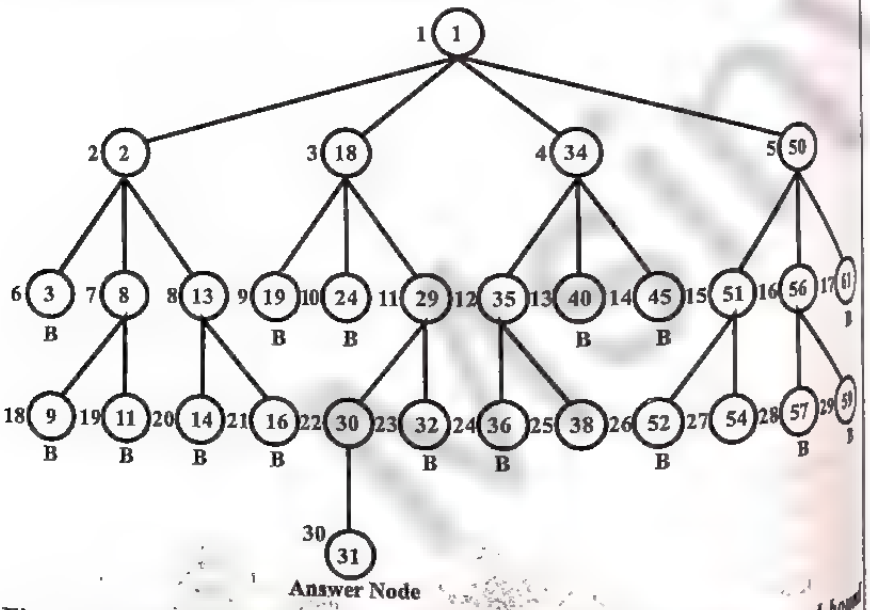


Fig. 4.19 4-queens State Space Tree Generated by FIFO Branch-and-bound

Here initially, there is only one live node, node 1. This is the case in which no queen has been placed on the chessboard. This node now becomes the E node. It is expanded and its children, nodes 2, 18, 34, and 50, are

generated. These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively. Now the only live nodes are 2, 18, 34, and 50. If the nodes generated are in this order, then the next E-node is node 2. It is also expanded and nodes 3, 18 and 13 are generated. Node 3 is immediately killed using the bounding function that no queens should attack and if we place 1st queen in row 1 and column 1, and place queen 2 in row 2 and column 2, the queen 1 and 2 result in same diagonal, so node 3 is killed immediately. Nodes 8 and 13 are added to the queue of live nodes. Node 18 becomes the next E-node. Now nodes 19, 24 and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding function. Node 29 is added to the queue of live nodes. Now the E-node is node 34.

Fig. 4.19 shows the portion of tree that is generated by a branch-and-bound search. Nodes that are killed are represented by "B" under them. Numbers outside the nodes give the order in which nodes are generated by FIFO branch-and-bound. At the time, the answer node i.e. node 13 is reached, the only nodes that are live are 38 and 54. But for above problem backtracking is better technique than branch-and-bound technique.

Q.25. Explain how branch and bound method can be used to solve any problem. What is least cost search ? ***(R.G.P.V., Dec. 2012)***

Ans. **Branch and Bound Method** – Refer to Q.24.

Least Cost Search – In least cost we give preference to an answer node with minimum cost. As in both the cases i.e., LIFO and FIFO, selection of next E-node is rigid and in a sense blind. It does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. As in fig. 4.19, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to an answer node in one move. However, the requirement of the rigid FIFO rule is the expansion of all live nodes generated before node 30 was expanded.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. Next E-node is selected according to this ranking function. If in 4-queen example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become the E-node following node 29. The remaining live nodes will never become E nodes since the expansion of node 30 results in the generation of an answer node (node 31).

The ideal way to assign ranks is according to the additional computational effort (or cost) required to reach an answer node from the live node. For any node x, this cost could be

(i) The number of nodes in the subtree x that should be generated before an answer node is generated or simply.

(ii) The number of levels the nearest answer node (in the subtree x) is from x.

Using cost measure (ii), the cost of the root of the tree of fig. 4.19 is 4 (node 31 is four levels from node 1). Similarly the costs of nodes 18 and 34, 29 and 35, and 30 and 38 are 3, 2 and 1 respectively. All remaining nodes on levels 2, 3, and 4 have costs respectively greater than 3, 2, and 1. Using these costs as a basis to select the next E-node, the E-nodes generated are nodes 2, 34, 50, 19, 24, 32 and 31. It should be noticed that if cost measure (i) is used, then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If the cost measure used is (ii), then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node. The major difficulty with using either of these ideal cost functions is that computing the cost of a node generally involves a search of the subtree x for an answer node. Thus, by the time the cost of a node is found, that subtree has been searched and there is no need to explore x again. This is the reason why search algorithms usually rank nodes only on the basis of an estimate $\hat{g}(\cdot)$ of their cost.

So, ranking function can be given as –

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

Here, h(x) is the cost of reaching x from the root and $f(\cdot)$ is an nondecreasing function and $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, to select the next E-node, would always choose for its next E-node a live node with least $\hat{c}(.)$. Hence, such a strategy is called an LC-search (Least-cost-search).

BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) - 0$ level of node x, then a LC-search generates nodes by levels. This is same as in BFS, if $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC branch-and-bound search.

Q.26. Explain how to solve sum of subset problem ?
(R.G.P.V., Dec. 2014)

Ans. In computer science, the subset sum problem is one of the important problem in complexity theory. The problem is given a set of integers, is there a non-empty subset whose sum is zero ? For example, given the set {–7, –3, –2, 5, 8}, the answer is yes because the subset (–3, –2, 5} sums to zero. The problem is NP-complete.

Q.27. Explain travelling salesperson problem. (R.G.P.V., Dec. 2008)

Ans. In the travelling-salesperson problem, we are given a complete undirected graph G = (V, E) that has a nonnegative integer cost c(u, v) associated with each edge (u, v) ∈ E and we must find a tour of G with minimum cost.

Let C(A) denotes the total cost of the edges in the subset $A \subseteq E$.

$$C(A) = \sum_{(u,v) \in A} c(u,v)$$

Practically, it is always cheapest to go directly from a place u to a place w, going by way of any intermediate stop v can't be less expensive. Or say, cutting out an intermediate stop never increases the cost. This can be formalized that the cost function c satisfies the triangle inequality, if for all vertices u, v, w ∈ V,

$$c(u,w) \leq c(u,v) + c(v,w)$$

This triangle inequality is natural one, and in many applications it is automatically satisfied. In this problem, our tour starts from an initial state and completes after returning to original state passing through all intermediate states.

If the graph has n vertices, i.e., |V| = n, then the solution space S is given by S = {1, π, 1, :π is a permutation of (2, 3, ..., n)}.

Then |S| = (n – 1)!

The size of S can be reduced by restricting S so that $(1, i_1, ..., i_2, i_{n-1}, 1) \in S$ iff $(i_j, i_j + 1) \in E$, $0 \leq j \leq n-1$, and $i_0 = i_n = 1$.

State space tree for this problem, for n = 4 and initial and final states 1 is shown in fig. 4.20.

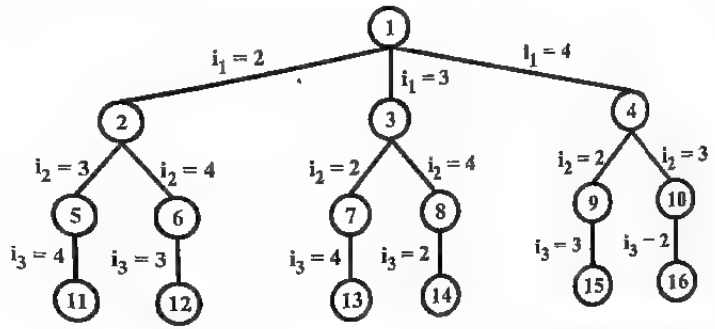


Fig. 4.20 State Space Tree for Travelling Salesperson Problem with n = 4 and $i_0 = i_4 = 1$

Q.28. Solve travelling salesman problem by using branch and bound technique with example. (R.G.P.V., Dec. 2010)

Or

What is branch and bound technique ? How travelling salesperson problem can be solved using this technique ? (R.G.P.V., Dec. 2011, June 2013)

Or

How can travelling salesperson problem be solved ? (R.G.P.V., June 2015)

Or

Apply the branch and bound algorithms to solve the travelling salesman problem. Use suitable graph. (R.G.P.V., May 2014)

Ans. **Branch and Bound Technique** – Refer to Q.23.

Travelling Salesman Problem – Least cost branch-and-bound (LCBB) technique can be used to find a minimum cost tour from a given set of the nodes. For this, we need to define two cost functions $\hat{c}(\cdot)$ and $u(\cdot)$ such that $\hat{c}(r) \le c(r) \le u(r)$ for all nodes r.

The cost function $c(\cdot)$ should be such that the solution node with least $c(\cdot)$ corresponds to a shortest tour in G. One way for deciding value of $c(\cdot)$ is –

C(A) = Length of tour defined by the path from the root to A, if A is a leaf cost of a minimum cost leaf in the subtree A, if A is not a leaf.

State space tree for such problem is shown in fig. 4.20 with each edge assigned a value indicates the root to be followed.

For example, node 8 shows the path 1, 3, 4 and node 13 shows path 1, 3, 2, 4. Reduced cost matrix is used for finding better value of $\hat{c}(\cdot)$. For this, we have to reduce a matrix. A matrix is said to be reduced if its rows and columns are reduced, and a row (column) is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.

We associate a reduced cost matrix with every node in the travelling salesperson state space tree. Suppose that A be the reduced cost matrix for a node P and Q be a child for node P, also suppose that P is not a leaf child. Here tree edge (P, Q) corresponds to including edge <i, j> in the tour.

A reduced cost matrix for P can be obtained as follows –

(i) Change all entries in row i and column j of A to ∞.

This prevents the use of any more edges leaving vertex i or entering vertex j.

(ii) Set A(j, 1) to ∞. This prevents the use of edge (j, 1).

(iii) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

Total cost of Q can be given as –

$$\hat{c}(Q) = \hat{c}(P) + A(i, j) + r$$

where r is the total amount subtracted in step (iii).

At each step of determining minimum cost tour we select a node having minimum cost and go ahead step-by-step by doing same, and finally we reach with a minimum cost tour.

Q.29. Explain travelling salesman problem using branch and bound method. Generate a state space tree for the following cost matrix –

$$C_{ij} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 12 & 7 & 4 \\ 10 & \infty & 13 & 9 \\ 3 & 8 & \infty & 11 \\ 5 & 6 & 10 & \infty \end{bmatrix} \end{array}$$

(R.G.P.V., June 2009, Dec. 2009)

Or

What do you understand by travelling salesman problem ? Discuss with suitable example. (R.G.P.V., June 2016)

Ans. **Travelling Salesman Problem** – Refer to Q.27.

Suppose the initial state is given by matrix –

$$\begin{bmatrix} \infty & 12 & 7 & 4 \\ 10 & \infty & 13 & 9 \\ 3 & 8 & \infty & 11 \\ 5 & 6 & 10 & \infty \end{bmatrix}$$

Fig. 4.21 (a) Initial State Cost Matrix

For finding the minimum cost tour we have to reduce the above matrix. For this we have to reduce each row and column of above matrix. Reduction can be done as follows –

(i) Subtract 4 from row one

(ii) Subtract 9 from row two

(iii) Subtract 3 from row three

(iv) Subtract 5 from row four.

$$\begin{bmatrix} \infty & 8 & 3 & 0 \\ 1 & \infty & 4 & 0 \\ 0 & 5 & \infty & 8 \\ 0 & 1 & 5 & \infty \end{bmatrix}$$

Fig. 4.21 (b) L = 21

Now, the above matrix is not reduced because it's all rows are contain at least one zero but not all column contain at least one zero. So, we further reduced it by setting at least one zero in all column of the above matrix. Cost of the above matrix is ; 4 + 9 + 3 + 5 = 21. By subtracting 1 from column 2 and subtracting 3 from column 3. The resulting matrix will be given as –

$$\begin{bmatrix} \infty & 7 & 0 & 0 \\ 1 & \infty & 1 & 0 \\ 0 & 4 & \infty & 8 \\ 0 & 0 & 2 & \infty \end{bmatrix}$$

Fig. 4.21 (c) Reduced Cost Matrix L = 25

Cost of the resulting matrix is given by adding cost of its original matrix i.e., 21 and the cost that is reduced from it i.e., 1 + 3, so, the cost of resulting matrix is; 21 + 1 + 3 = 25. Hence, all tours in the original graph have a length at least 25.

For this, we can show a tree representation of state space generated by least cost branch and bound (LCBB) as shown in fig. 4.22. Starting with root node as the E-node, nodes 2, 3 and 4 generated. In tree nodes show initial position of tour with minimum cost 25 as obtained by the matrix of fig. 4.21(c). The reduced matrix for these nodes i.e., 2, 3 and 4 can be obtained in the following manner –

Reduced cost matrix for node 2 i.e., path (1, 2) will be obtained by –

(i) Setting all entries in row 1 to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 1 & \infty & 1 & 0 \\ 0 & 4 & \infty & 8 \\ 0 & 0 & 2 & \infty \end{bmatrix}$$

Fig. 4.21 (d)

(ii) Setting all entries in column 2 to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 1 & \infty & 1 & 0 \\ 0 & \infty & \infty & 8 \\ 0 & \infty & 2 & \infty \end{bmatrix}$$

Fig. 4.21 (e)

(iii) Setting entry (2, 1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & 0 \\ 0 & \infty & \infty & 8 \\ 0 & \infty & 2 & \infty \end{bmatrix}$$

Fig. 4.21 (f)

(iv) Reducing the cost of matrix by analyzing matrix of fig. 4.21(f). We see that it is not in reduced form. So, we reduced this matrix by setting at least one zero in column 3. The resultant matrix is –

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ 0 & \infty & \infty & 8 \\ 0 & \infty & 1 & \infty \end{bmatrix}$$

Fig. 4.21 (g) L = 33

(v) Cost of the matrix can be find by the formula –

$$C(Q) = C(P) + A(i, j) + r$$

where, C(Q) is the required cost, C(P) is the cost of original matrix, A(i, j) is the cost of (i, j) entry in matrix A, r is the total cost reduced.

$$C(2) = C(1) + A(1, 2) + r$$
$$= 25 + 7 + 1 = 33$$

Similarly, reduced cost matrix for node 3 i.e., Path (1, 3) will be obtained by –

(i) Setting all entries in row 1 to ∞

(ii) Setting all entries in column 3 to ∞

(iii) Setting entry (3, 1) to ∞

(iv) Reduce the resulting matrix

(v) Find the cost of the matrix.

$$C(3) = C(1) + A(1, 3) + r$$
$$= 25 + 0 + 4$$
$$= 29$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 0 \\ \infty & 0 & \infty & 4 \\ 0 & 0 & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (h) L = 29

Reduced cost matrix for node 4 i.e., path (1, 4) will be obtained by –

(i) Setting all entries in row 1 to ∞

(ii) Setting all entries in column 4 to ∞

(iii) Setting entry (4, 1) to ∞

(iv) Reduced the cost matrix

(v) Find the cost of the matrix.

$$C(4) = C(1) + A(1, 4) + r$$
$$= 25 + 0 + 1$$
$$= 26$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 2 & \infty \end{bmatrix}$$

Fig. 4.21 (i) L = 26

Now, we choose the path with minimum cost, i.e., node 4 or path (1, 4) and find next possible nodes of 4 i.e., 5 and 6. First, computing the path (1, 4, 2).

For this, we will use node 4 and resultant matrix with minimum cost. For this following steps will be followed –

(i) Setting all entries of row 4 to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (j)

(ii) Setting all entries in column 2 to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (k)

(iii) Setting entry (2, 1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (l)

(iv) Reducing the cost of matrix of fig. 4.21 (*l*) by setting at least one zero in all rows and all column.

(v) Cost of the reduced matrix is –

$$C(5) = C(4) + A(4, 2) + r$$
$$= 26 + 0 + 0 = 26$$

Now, we compute the path (1, 4, 3) for node 6 with the following steps-

(i) Setting all entries of row 4 to ∞

(ii) Setting all entries of column 3 to ∞

(iii) Setting entry (3, 1) to ∞

(iv) Reduced the cost of matrix

(v) Find the cost of matrix.

$$C(6) = C(4) + A(4, 3) + r$$
$$= 26 + 2 + 4 = 32$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (m) L = 32

Now, we proceed with node 5 having minimum cost 26. The path till now is 1, 4, 2. The last choice we have is 3 i.e., $i_3 = 3$. The following sequence of steps is to be followed –

(i) Setting all entries of row 2 to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (n)

(ii) Setting all entries of column 3 to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (o)

(iii) Setting entry (3, 1) to ∞, which is already exist.

(iv) Reducing the cost of matrix of fig. 4.21 (o) by analyzing at least one zero in all rows and columns

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.21 (p) L = 28

(v) Cost of the resultant matrix is –

$$C(7) = C(5) + A(2, 3) + r$$
$$C(7) = 26 + 0 + 0 = 26$$

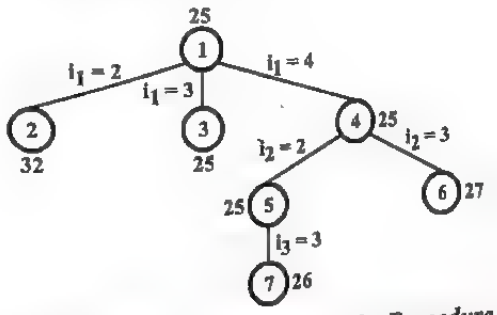


Fig. 4.22 State Space Tree Generated by Procedure LCBB

We know that, in travelling salesman problem initial and final positions are same. Therefore, path is (1, 4, 2, 3, 1) with minimum cost 26.

Q.30. What is branch and bound method, using branch and bound method generate a state space tree for the following cost matrix.

$$C_{ij} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 12 & 7 & 4 \\ 10 & \infty & 13 & 9 \\ 3 & 8 & \infty & 11 \\ 5 & 6 & 10 & \infty \end{bmatrix} \end{array}$$

(R.G.P.V., Dec. 2013)

Ans. Refer to Q.23 and Q.29.

NUMERICAL PROBLEMS

Prob.3. Consider the travelling salesperson instance defined by the cost matrix –

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

(i) Obtain the reduced cost matrix

(ii) Using LCBB find least cost path.

Or

Consider the travelling salesperson instance defined by the cost matrix-

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

Obtain the reduced cost matrix and solve it.

(R.G.P.V., Dec. 2007, June 2014, Dec. 2014, 2015)

Sol. For finding reduced cost matrix of above matrix, first we try to reduce its rows.

Subtracting minimum cost from corresponding row.

(i) Subtract 3 from R_1 (ii) Subtract 3 from R_2

(iii) Subtract 5 from R_3 (iv) Subtract 3 from R_4

(v) Subtract 8 from R_5 .

Cost of matrix is $(3 + 3 + 5 + 3 + 8) = 22$.

Resulting matrix is shown as below –

$$\begin{bmatrix} \infty & 4 & 0 & 9 & 5 \\ 0 & \infty & 3 & 11 & 6 \\ 0 & 3 & \infty & 1 & 13 \\ 6 & 0 & 2 & \infty & 8 \\ 10 & 6 & 1 & 0 & \infty \end{bmatrix} \quad \begin{array}{l} R_1 = R_1 - 3 \\ R_2 = R_2 - 3 \\ R_3 = R_3 - 5 \\ R_4 = R_4 - 3 \\ R_5 = R_5 - 8 \end{array}$$

Fig. 4.23 Cost = 22 for Finding Reduced Cost Matrix

Here, we see that each row in resulting matrix has a zero entry. Also, all the columns have a zero entry except last (C_5) column –

So subtracting 5 from last column

$$C_5 = C_5 - 5$$

The resulting matrix is –

$$\begin{bmatrix} \infty & 4 & 0 & 9 & 0 \\ 0 & \infty & 3 & 11 & 1 \\ 0 & 3 & \infty & 1 & 8 \\ 6 & 0 & 2 & \infty & 3 \\ 10 & 6 & 1 & 0 & \infty \end{bmatrix}$$

Fig. 4.24 Cost = 27 Reduced Cost Matrix

Total cost of resulting matrix is total value deducted, i.e.,

$$22 + 5 = 27.$$

Now, above matrix has one zero entry in each row and column so, this matrix is the required reduced cost matrix. With cost 27, it indicates that minimum cost path has at least cost 27.

Suppose, we start travelling from node 1 having cost 27 as shown in fig. 4.25.

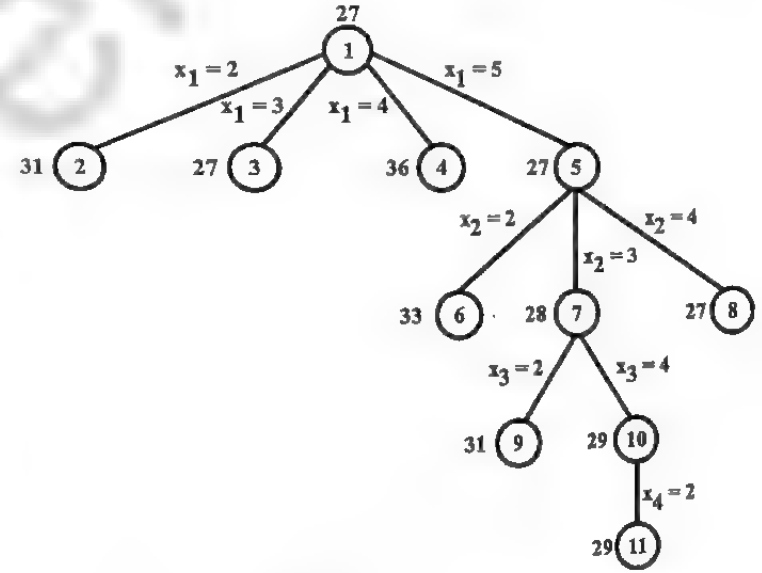


Fig. 4.25 State Space Tree Generated By LCBB

Further we try to find the matrix for path (1,2) i.e., $x_1 = 2$.

Steps to be followed are –

(i) Setting all the entries in the row 1 and column 2 to ∞

(ii) Setting entry (2,1) to ∞

(iii) Reducing the resulting matrix

(iv) Find the cost of matrix.

For (i) and (ii) we consider the matrix of fig. 4.24 as root matrix, resulting matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & 11 & 1 \\ 0 & \infty & \infty & 1 & 8 \\ 6 & \infty & 2 & \infty & 3 \\ 10 & \infty & 1 & 0 & \infty \end{bmatrix}$$

Fig. 4.26 All Entries of Row 1 and Column 2 to ∞ also set (2,1) to ∞

(iii) We see that resulting matrix of fig. 4.26 is not reduced, so reducing it we get

(a) Subtract 1 from R_2 (b) Subtract 2 from R_4.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & 10 & 0 \\ 0 & \infty & \infty & 1 & 8 \\ 4 & \infty & 0 & \infty & 1 \\ 10 & \infty & 1 & 0 & \infty \end{bmatrix} \quad \begin{matrix} R_2 = R_2 - 1 \\ \\ R_4 = R_4 - 2 \end{matrix}$$

Fig. 4.27 For Path (1,2)

Here, we see that all rows and columns have at least one zero entry, so resulting matrix is reduced.

(iv) Total cost is

$$C(2) = C(1) + A(1,2) + r$$
$$C(2) = 27 + 4 + 3 = 34$$

This is shown in fig. 4.25 by one edge (1,2) having cost 34.

In similar fashion we try to find path (1,3), i.e., for $x_1 = 3$. Steps to be followed are –

(i) Make all the entries in row 1 and column 3 to ∞ and set entry (3,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost of matrix.

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 11 & 1 \\ \infty & 3 & \infty & 1 & 8 \\ 6 & 0 & \infty & \infty & 3 \\ 10 & 6 & \infty & 0 & \infty \end{bmatrix}$$

Fig. 4.28 For Path (1,3)

(ii) For reducing we subtract 1 from third row.

The resulting matrix is –

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 11 & 1 \\ \infty & 2 & \infty & 0 & 7 \\ 6 & 0 & \infty & \infty & 3 \\ 10 & 6 & \infty & 0 & \infty \end{bmatrix} \quad R_3 = R_3 - 1$$

Fig. 4.29 For Path (1,3) Cost = 28

But resulting matrix is not reduced as column 5 does not have any zero entry. So, $(R_5 = R_5 - 1)$ i.e., reducing 1 from column 5.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 11 & 0 \\ \infty & 2 & \infty & 0 & 6 \\ 6 & 0 & \infty & \infty & 2 \\ 10 & 6 & \infty & 0 & \infty \end{bmatrix} \quad R_5 = R_5 - 1$$

Fig. 4.30 For Path (1,3) Cost = 29

(iii) Total cost of this matrix is –

$$C(3) = C(1) + A(1,3) + r$$
$$= 27 + 0 + 2 = 29$$

This is shown in fig. 4.25 by edge (1,3) and cost 29.

Further we try to find for path (1,4), i.e., $x_1 = 4$.

Steps to be followed are –

(i) Set all the entries in row 1 and column 4 to ∞ and set entry (4,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost of matrix.

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty & 1 \\ 0 & 3 & \infty & \infty & 8 \\ \infty & 0 & 2 & \infty & 3 \\ 10 & 6 & 1 & \infty & \infty \end{bmatrix}$$

Fig. 4.31 For Path (1,4)

(ii) Reducing it –

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty & 0 \\ 0 & 3 & \infty & \infty & 7 \\ \infty & 0 & 2 & \infty & 2 \\ 9 & 5 & 0 & \infty & \infty \end{bmatrix} \quad \begin{matrix} R_5 = R_5 - 1 \\ C_5 = C_5 - 1 \end{matrix}$$

Fig. 4.32 For Path (1,4)

(iii) Cost of this matrix is –

$$C(4) = C(1) + A(1,4) + r$$
$$= 27 + 9 + 2 = 38$$

It is shown in fig. 4.25 by edge (1, 4) and cost 38.

For finding matrix of path (1,5) we do

(i) Set all the entries in row 1 and column 5 to ∞ and set entry (5,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost of matrix.

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & 11 & \infty \\ 0 & 3 & \infty & 1 & \infty \\ 6 & 0 & 2 & \infty & \infty \\ \infty & 6 & 1 & 0 & \infty \end{bmatrix}$$

Fig. 4.33 For Path (1,5)

(ii) For reducing we subtract 1 from column 3 so,

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 2 & 11 & \infty \\ 0 & 3 & \infty & 1 & \infty \\ 6 & 0 & 1 & \infty & \infty \\ \infty & 6 & 0 & 0 & \infty \end{bmatrix} \quad C_3 = C_3 - 1$$

Fig. 4.34 For Path (1,5)

(iii) Cost $C(5) = C(1) + A(1,5) + r = 27 + 0 + 1 = 28$

It is shown in fig. 4.25 by edge (1, 5) and cost 28.

Further, we go ahead by node of minimum cost, i.e., node 5 of cost 28.

Further possible values of x_2 are 2, 3 and 4 so, if $x_2 = 2$, i.e., we try for path 1, 5, 2

Steps to be followed are –

(i) Set all the entries in row 5 and column 2 to ∞ and set entry (2,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost.

(i) For this, we consider matrix of fig. 4.34 as root matrix –

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & 11 & \infty \\ 0 & \infty & \infty & 1 & \infty \\ 6 & \infty & 1 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.35 For Path (1,5,2)

(ii) Above matrix is not reduced so, subtracting 2 from second row and 1 from 4th row –

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 9 & \infty \\ 0 & \infty & \infty & 1 & \infty \\ 5 & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \quad \begin{matrix} R_2 = R_2 - 2 \\ R_4 = R_4 - 1 \end{matrix}$$

Fig. 4.36 For Path (1,5,2)

Again reducing 1 from column 4 –

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 8 & \infty \\ 0 & \infty & \infty & 0 & \infty \\ 5 & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \quad C_4 = C_4 - 1$$

Fig. 4.37 For Path (1,5,2)

(iii) Total cost is $C(6) = C(5) + A(5,2) + r$
$= 28 + 6 + 4 = 38$

For finding path of (1, 5, 3), i.e., $x_2 = 3$ –

(i) Make row 5 and column 3 to ∞ and make entry (3,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost.

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 11 & \infty \\ \infty & 3 & \infty & 1 & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.38 For Path (1,5,3)

(ii) For reducing it we subtract 1 from row 3.

Resulting matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 11 & \infty \\ \infty & 2 & \infty & 0 & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \quad R_3 = R_3 - 1$$

Fig. 4.39 For Path (1,5,3)

This matrix is reduced.

(iii) Cost is $C(7) = C(5) + A(5,3) + r$
$= 28 + 0 + 1 = 29$

It is shown in fig. 4.25 by edge (1, 5, 7).

Further we try for path (1, 5, 4), steps to be followed are –

(i) Make row 5 and column 4 to ∞ and make entry (4,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost of matrix.

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 2 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.40 For Path (1,5,4)

(ii) For reducing it, we subtract 1 from column 3.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \quad C_3 = C_3 - 1$$

Fig. 4.41 For Path (1,5,4)

(iii) For finding total cost –

$$C(8) = C(5) + A(5,4) + r = 28 + 0 + 1 = 29$$

It is shown in fig. 4.25 by edge (1, 5, 8).

Further we continue with node of minimum cost, i.e., node 7 of cost 29.

The next possible value for x_3 are 2 and 4, i.e., $x_3 = 2$ or $x_3 = 4$.

we try for $x_3 = 2$, i.e., for path (1,5,3,2).

For this we consider matrix of fig. 4.39 as root matrix.

Steps to be followed are –

(i) Make the entries of row 3 and column 2 as ∞ and make entry (2,1) to ∞

(ii) Reduce the matrix (iii) Find the cost.

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 11 & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 6 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.42 For Path (1,5,3,2)

(ii) For reducing it, we subtract 11 from row second and 6 from row fourth, resulting matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \quad R_2 = R_2 - 1$$

Fig. 4.43 For Path (1,5,3,2)

(iii) This matrix is reduced for finding cost

$$C(9) = C(7) + A(3,2) + r = 29 + 2 + 17 = 48$$

It is shown in fig. 4.25 by edge (1, 5, 7, 9)

Further we try for path (1,5,3,4) i.e. $x_3 = 4$

Steps to be followed are –

(i) Make the entries of row 3 and column 4 as ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.44 For Path (1,5,3,4)

(ii) Make entry (4,1) to ∞. The resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.45 For Path (1,5,3,4)

(iii) The matrix is already in reduced form.

(iv) For finding total cost

$$C(10) = C(7) + A(3,4) + r = 29 + 0 + 0 = 29$$

It is shown in fig. 4.25 by edge (1, 5, 7, 10).

Now, we proceed with node having minimum cost. So, we proceed with node 10 having cost 29.

The path known till now is 1,5,3,4.

The last choice we have is 2 i.e., $x_4 = 2$

(i) Make row 4 and column 2 as ∞ and make entry (2,1) to ∞

(ii) Reduce the matrix

(iii) Find the cost of matrix.

i.e., for path (1,5,3,4,2) –

(i)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 4.46 For Path (1,5,3,4,2)

(ii) Above matrix is already reduced, so no changes are needed.

(iii) Total cost can be found as –

$$C(11) = C(10) + A(4,2) + r = 29 + 0 + 0 = 29$$

It is shown in fig. 4.25 by edge (1, 5, 7, 10, 11).

So, now we know that in travelling salesperson problem initial and final position are same so, path of minimum cost is (1,5,3,4,2,1) with minimum cost of 29.

Prob.4. Solve the TSP using branch and bound technique –

	A	*B*	*C*
A	∞	*2*	*3*
B	*5*	∞	*3*
C	*2*	*4*	∞

(R.G.P.V., June 2017)

Sol. First convert the given matrix into reduced cost matrix

	A	B	C
A	∞	2	3
B	5	∞	3
C	2	4	∞

(i) Subtract 2 from row 1
(ii) Subtract 3 from row 2
(iii) Subtract 2 from row 3

Resultant matrix

	A	B	C
A	∞	0	1
B	2	∞	0
C	0	2	∞

Cost of the above matrix is the sum of the total cost reduced from the original matrix.

i.e., $2 + 3 + 2 = 7$

The resultant matrix is reduced matrix because its rows and columns have atleast one zero value.

Now generate the state space tree.

Hence, Path = A → B → C → A

Cost = 2 + 3 + 2 = 7

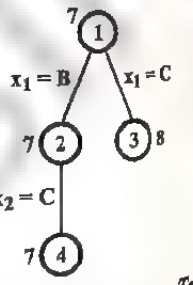


Fig. 4.47 State Space Tree

Prob.5. Consider the following initial cost matrix for traveling salesman problem. Solve this problem using branch and bound technique.

$$\begin{bmatrix} \infty & 4 & 5 \\ 7 & \infty & 5 \\ 4 & 6 & \infty \end{bmatrix}$$

(R.G.P.V., Nov. 2018)

Sol. First convert the given matrix into reduced cost matrix.

	A	B	C
A	∞	4	5
B	7	∞	5
C	4	6	∞

(i) Subtract 4 from row 1

(ii) Subtract 5 from row 2

(iii) Subtract 4 from row 3

Resultant matrix

	A	B	C
A	∞	0	1
B	2	∞	0
C	0	2	∞

Cost of the above matrix is the sum of the total cost reduced from the original matrix.

i.e. $4 + 5 + 4 = 13$

The resultant matrix is reduced matrix because its rows and columns have atleast one zero value.

Hence, Path = A → B → C → A

Cost = 4 + 5 + 4 = 13

Prob.6. Solve the following instance of the Knapsack problem by the branch and bound algorithm.

Item	*Weight*	*Value*	*Value / Weight*
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

The Knapsack capacity m is 10. ***(R.G.P.V., Dec. 2016)***

Sol. Fig. 4.48 shows the solution of given knaps

Hence the result is

$(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ **Ans.**

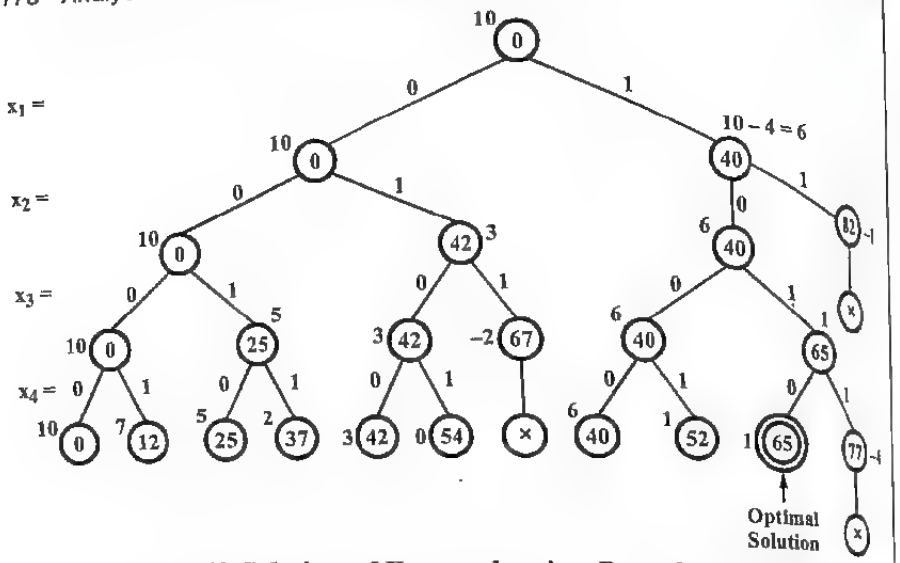


Fig. 4.48 Solution of Knapsack using Branch and Bound

Prob.7. Draw the portion of state space tree generated by LC branch and bound for the following Knapsack instance n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18); (w_1, w_2, w_3, w_4) = (2, 4, 6, 9) and M = 15.

(R.G.P.V., June 2014, Dec. 2014)

Sol. Let us trace the working of an LC branch-and-bound search using $\hat{C}(.)$ and u(.). We continue to use the fixed tuple size formulation – The search begins with the root as the E-node. For this node, node L of fig. 4.49, We have $\hat{C}(1) = -38$ and $\mu(1) = -32$.

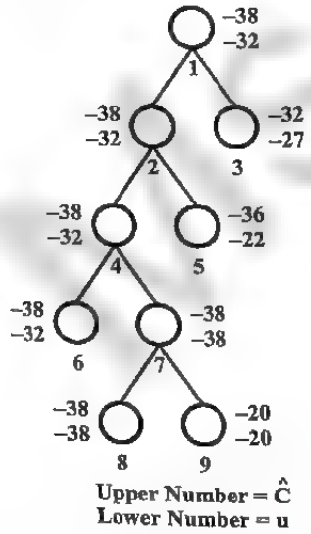


Fig. 4.49 LC Branch-and-bound Tree

The computation of u(1) and $\hat{C}(1)$ is done as follows. The bound u(1) has a value U Bound (0, 0, 0, 15). U Bound scans through the objects from left to right starting from j. It adds these objects into the Knapsack until the first object that does not fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus Cw is returned. In function UBound C and b start with a value of zero. For i = 1, 2 and 3, C gets incremented by 2, 4 and 6 respectively. When i = 4, the test $(C + w[i] \le m)$ fails and hence the value returned is – 32. Function Bound is similar to UBound, except that, it also considers a fraction of the first object that does not fit the Knapsack. For example, in computing $\hat{C}(1)$, the first object that does not fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction 3/9 of the object 4 and hence returns

$$-32 - \frac{3}{9} * 18 = -38.$$

Since node 1 is not a solution node, LCBB sets ans = 0 and upper = –32. The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{C}(2) = -38$, $\hat{C}(3) = -32$, u(2) = – 32, and u (3) = – 27. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node 4 is the live node with least $\hat{C}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and upper is updated to – 38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then upper is updated to – 38 and node 8 is put onto the live nodes list. Node 9 has $\hat{C}(9) >$ upper and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{C}$. Regardless of which becomes the next E–node, $\hat{C}(E) \ge$ upper and the search terminates with node 8 the answer node. At this time, the value – 38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\Sigma p_i x_i$ = upper. Hence a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, tag. The sequence of tag bits from the answer node to the root give the x_i values. Thus, we have tag (2) = tag (4) = tag (6) = tag (8) = 1 and tag (3) = tag (5) = tag (7) = tag (9) = 0. The tag sequence for the path 8, 7, 4, 2, 1 is 1011 and so $x_4 = 1$, $x_3 = 0$, $x_2 = 1$, and $x_1 = 1$.

MEANING OF LOWER BOUND THEORY AND ITS USE IN SOLVING ALGEBRAIC PROBLEM, INTRODUCTION TO PARALLEL ALGORITHMS

Q.31. Explain the term lower bound with suitable example.

(R.G.P.V., Dec. 2015)

Ans. Lower bound is an estimate of a number of operations needed to solve a given problem. If f(n) is the time for some algorithm, then we write $f(n) = \Omega(g(n))$ to mean that g (n) is a lower bound for f (n). Formally, this equation can be written if there exist positive constants c and n_o such that $|f(n)| \geq c|g(n)|$ for all $n > n_o$. For example, in sorting (comparison-based) and searching in a sorted array the lower bound is Ω (n log n) and Ω (log n) respectively.

Q.32. Explain lower bound theory and its use in solving algebraic problems. ***(R.G.P.V., June 2009, Dec. 2009, June 2010)***

Or

Write a short note on lower bound theory. ***(R.G.P.V., Dec. 2010)***

Or

What is the meaning of lower bound theory and how can it be used in solving algebraic problems ? ***(R.G.P.V., May 2019)***

Ans. Lower bound theory is a technique that has been used to establish that a given algorithm is the most efficient possible. This is done by discovering a function g(n) that is a lower bound on the time that *any* algorithm must take to solve the given problem. Now if we have an algorithm whose computing time is the same order as g(n), then we know that asymptotically we can do no better.

If f(n) is the time for some algorithm, then we write $f(n) = \Omega(g(n))$ to mean that g(n) is a lowerbound of f(n). This equation can be written formally, if there exists positive constants c and n_0 such that $|f(n)| \geq c\,|g(n)|$ for all $n > n_0$. In addition to developing lower bounds to within a constant factor, we are more concious of the fact to determine more exact bounds whenever this is possible.

Deriving good lowerbounds is more difficult than devising efficient algorithms. This happens because a lower bound states a fact about *all* possible algorithms for solving a problem. Generally we cannot enumerate and analyse all these algorithms, so lowerbound proofs are often hard to obtain.

The proof techniques that are useful for obtaining lower bounds are –

(i) Comparison Trees – Comparison trees are the computational model useful for determining lower bounds for sorting and searching problems.

(ii) Oracles and Adversary Arguments – One of the proof technique that is important for obtaining lower bounds consists of making use of an

oracle. The most famous oracle in history was called the Delphic oracle, located in Delphi, Greece. Still this oracle can be found situated in the side of a hill embedded in some rocks. In olden times people come across oracle and ask it a question. After some period of time elapsed, the oracle would reply and caretaker would interpret the oracle reply.

Similar phenomenon happens when we use an oracle to establish lower bounds. Given some model of computation such as comparison trees, the oracle tells us the outcome of each comparison. For getting a good lower bound, the oracle tries its best to cause the algorithm to work as hard as it can. It does this by choosing as the outcome of the next test, the result that causes the most work to be required to find the final answer. Also if keeps track of the work that is done, a worst-case lower bound for the problem can be derived.

(iii) Lower Bounds through Reductions – This is a very important technique for getting lower bounds. This technique calls for *reducing* the given problem to another problem for which a lower bound is already known. We define it as follows –

Let us consider that P_1 and P_2 be any two problems. We say P_1 reduces to P_2 (also written $P_1 \propto P_2$) in time *T(n)* if an instance of P_1 can be converted into an instance of P_2 and a solution for P_1 can be obtained from a solution for P_2 in time $\leq T(n)$.

Let us take an example to understand it. Suppose P_1 be the problem of selection and P_2 be the problem of sorting. Suppose the input have n numbers. If the numbers are sorted, say in an array A[ ], the ith – smallest element of the input can be obtained as A[i]. Hence P_1 reduces to P_2 in O(1) time.

It should be noted that if P_1 reduces to P_2 in *t(n)* time and if T(n) be the lower bound on the solution time of P_1, then, clearly, T(n) – *t(n)* is a lower bound on the solution time P_2.

(iv) Techniques for Algebraic Problems – *Substitution* and *linear independence* are two methods used for deriving lower bounds on algebraic and arithmetic problems. The algebraic problems are operations on integers, polynomials and rational function.

Straight-line program is the model used for computation. It is called so because there are no branching instructions allowed. This means that if we know a way of solving a problem for n inputs, then a set of straight-line programs, one each for solving a different size n, can be given. The only statement used in straight-line program is the assignment which has the form s := p **op** q; where s, p, q are variables of bounded size and **op** is typically one of the arithmetic operations – addition, subtraction, multiplication, or division. Moreover p and q are either constants, input variables, or variables that have already appeared on the left of an assignment statement.

Consider, for example, one possible straight-line program that computes the value of a degree-two polynomial $a_2x^2 + a_1x + a_0$ has the form

$$V_1 := a_2 * x;$$
$$V_1 := V_1 + a_1 ;$$
$$V_1 := V_1 * x;$$
$$ans := V_1 + a_0;$$

Now to calculate the complexity of a straight-line program, we assume that each instruction takes one unit of time and requires one unit of space. Then the time complexity of a straight-line program is its number of assignments or its length. A more accurate assumption takes care of the fact than an integer n requires $\lfloor \log n \rfloor + 1$ bits to represent it. Since here operands are small, unit-cost assumption is appropriate.

Q.33. Discuss parallel algorithm briefly. ***(R.G.P.V., June 2011)***

Or

Explain parallel algorithms with suitable examples. ***(R.G.P.V., Dec. 2012)***

Or

Write about parallel algorithms with example. ***(R.G.P.V., Dec. 2013)***

Or

Write short note on parallel algorithm. ***(R.G.P.V., June 2017, Dec. 2017, May 2018, Nov. 2018)***

Or

Write a detailed note on parallel algorithms. ***(R.G.P.V., May 2019)***

Ans. Parallel machines are computers with more than one processor. Even the fastest single-processor machines may not be able to come up with solutions within tolerable time limits. Parallel machines offer the potential of decreasing the solution times enormously.

The idea of parallel computing is very simple. Given a problem to solve, we partition the problem into many subproblems; let each processor work on a subproblem; and when all the processors are done, the partial solutions are combined to arrive at the final answer.

RAM (Random Access Machine) is the sequential computational model. In RAM model we assume that any of the following operations can be done in one unit of time – addition, subtraction, multiplication, division, comparison, memory access, assignment and so on. This model is widely accepted as a valid sequential model. On the other hand when it comes to parallel computing, numerous models have been proposed and algorithms have been designed for each such models.

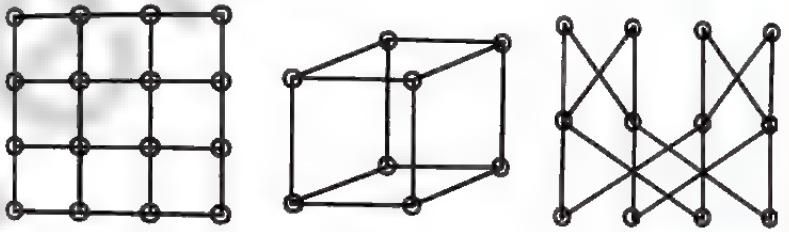
A very important feature of parallel computing that is absent in sequential computing is the requirement for interprocessor communication. For example,

given any problem, the processors have to communicate among themselves and agree on the subproblems each with work on. Moreover, they also require to communicate to see whether every one finished its task and so on. Each machine or processor in a parallel computer can be assumed to be RAM. Many parallel models differ in the way they support interprocessor communication. Broadly, parallel models can be divided into two –

(i) Fixed connection machines and

(ii) Shared memory machines.

A ***fixed connection network*** is a graph G(V, E) whose nodes represent processors and whose edges represent communication links between processor. Here, the degree of each node is assumed to be either a constant, or a slowly increasing function of the number of nodes in the graph. Examples include the mesh, hypercube, butterfly and so on. Fig. 4.50 shows the examples of fixed connection machines.



(a) Mesh *(b) Hypercube* *(c) Butterfly*

Fig. 4.50 Examples of Fixed Connection Machines

Here, in this model, interprocessor communication is done through the communication links. Any of the two processors connected by an edge in G can communicate in one step. Generally two processors can communicate through any of the paths connecting them. The communication time that is needed, depends upon the lengths of these paths (at least for small packets).

In ***shared memory models*** [also called PRAMs (Parallel Random Access Machines)], a number (say p) of processors work synchronously. They communicate with each other by using a common block of global memory that is accessible by all. This global memory is known as ***common*** or ***shared*** memory. Fig 4.51 shows a parallel random access machine.

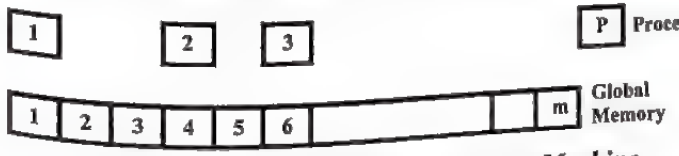


Fig. 4.51 A Parallel Random Access Machine

Here communication is done by writing to and/or reading from the common memory. Any two processors i and j can communicate in two steps–

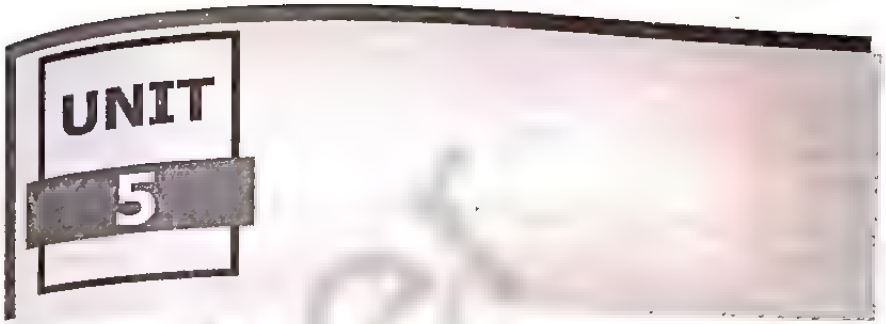
(i) Processor i writes its message into memory cell j

(ii) Processor j reads from this cell.

However, in a fixed connection machine, the communication time depends on the lengths of the paths connecting the communicating processors. Each processor in a PRAM is a RAM with some local memory. A single step of a PRAM algorithm can be one of the following – comparison, assignment, arithmetic operation (such as addition, division, and so on), memory access (local or global), etc. Typically, the number of cells (m) in the global memory is assumed to be the same as p. However this is not always the case. Infact there are algorithms for which m is much larger or smaller than p. Here we assume that input is given in the global memory and there is space for the output and for storing intermediate results. As the global memory is accessible by all processors, access conflicts may arise. However what happens if more than one processor tries to access the same global memory cell (for the purpose of reading from or writing into) ? There are many ways of resolving read and write conflicts. Accordingly, many variants of the PRAM arise.

EREW (Exclusive Read and Exclusive Write) PRAM is the shared memory model in which no concurrent read or write is allowed on any cell of the global memory. Here point to be noted is that ER and EW do not preclude different processors simultaneously accessing different memory cells. CREW (Concurrent Read and Exclusive Write) PARAM is a slight variation that allows concurrent reads but not concurrent writes. Similarly ERCW model is defined. Finally, the CRCW PARAM model permits both concurrent reads and concurrent writes.

★★



BINARY SEARCH TREES, HEIGHT BALANCED TREES, 2-3 TREES, B-TREES

Q.1. Explain binary search tree. List out its properties. ***(R.G.P.V., Dec. 2014)***

Or

Define binary tree. ***(R.G.P.V., June 2015)***

Or

Write the rules to construct binary search tree. ***(R.G.P.V., June 2016)***

Or

Write short note on binary search tree. ***(R.G.P.V., Dec. 2017)***

Ans. A ***binary search tree*** is a binary tree that is either empty or in which every node contains a key and satisfies the following properties –

(i) The key in the left child of a node (if it exists) is less than the key in its parent node.

(ii) The key in the right child of a node (if it exists) is greater than the key in its parent node.

(iii) The left and right subtrees of the root are again binary search trees.

(iv) Every element has a key and no two elements have the same key.

Q.2. What is binary search tree ? Give a comparison of binary search tree and heap. ***(R.G.P.V., Dec. 2012)***

Ans. **Binary Search Tree** – Refer to Q.1.

Comparison of Binary Search Tree and Heap – The definition of a heap excludes them from being binary search trees because in a heap, a node's right child cannot be greater than its parent, which is a necessary condition for a binary search tree.

The binary search tree looks good, if it is balanced. However, a binary search tree can become skewed, which reduces the efficiency of the operations. The heap, on the other hand, is always a tree of minimum height. The heap is

not a good structure for accessing any randomly selected element, but that is not one of the operations defined for priority queues. The accessing function of a priority queue specifies that only the largest element can be accessed. For the operations specified for priority queues, therefore, the heap is an excellent choice.

Q.3. How search can be applied on a binary search tree ? Explain with example.

Or

Write an algorithm to search a binary search tree T for an identified X. Assume that each node in T has three fields LCHILD, DATA and RCHILD. What is the computing time of your algorithm. ***(R.G.P.V., Dec. 2006)***

Or

For a given binary tree, design an algorithm to verify it is a binary search tree or not. Also find the complexity of your algorithms. ***(R.G.P.V., June 2012)***

Ans. The most common operation performed on a binary search tree is searching for a key stored in a tree. We use the algorithm 5.1 to search for a node with a given key in a binary search tree. We have been given a pointer to the root of the tree and a key k. TREE-SEARCH returns a pointer to a node with key k if one exists, otherwise, it returns NIL.

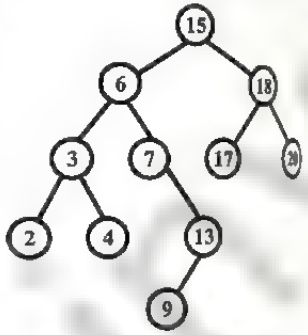


Fig. 5.1 Binary Search Tree

Algorithm 5.1

TREE-SEARCH (x, k)

1. **if** x = NIL or k = key [x]
2. **then return** x
3. if k < key [x]
4. **then return** TREE-SEARCH ((left [x], k)
5. **else return** TREE-SEARCH (right [x], k)

For example, consider the fig. 5.1.

The procedure begins its search at the root and then traces a path downward in the tree as shown in fig. 5.1. For each node x, that it reaches, it compares the key k with **key [x].** If the two keys are equal, the search terminates. If k is smaller than key [x], the search continues in the left subtree of x, as the binary-search-tree property means that k could not be stored in the right subtree. Symmetrically, if k is larger than key [x], the search continues in the right subtree.

For example, see fig. 5.2.

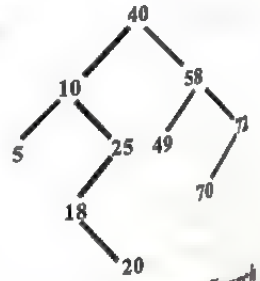


Fig. 5.2 A Binary Search Tree and Search Item 18

(i) For searching item 18 in above tree first we compare it with root item i.e., 40, 18 < 40.

(ii) So, we have to search left subtree of root element.

(iii) Comparing 18 with 10 here 10 < 18, so search in right subtree of element 10.

(iv) Compare 18 with 25. 18 < 25 so, 18 is there, in left subtree of element 25.

(v) Compare 18 with left subtree of element 25, i.e. 18. So, here we get the require position.

Path of search is shown by shaded area in fig. 5.2.

Computing Time of Algorithm – A binary search tree of height h can be searched by key as well as the rank in O(h) time.

Q.4. Write an algorithm to insert an item into a binary search tree and find the complexity of this algorithm.

Ans. Another important operation is to create and maintain a binary search tree. While inserting any node, we should take care that the resulting tree satisfies the properties of a binary search tree. A new node will always be inserted at its correct position in the binary search tree as a leaf.

For example, if we have following set of items to be inserted into a binary search tree i.e.

10, 15, 12, 7, 8, 18, 6, 20

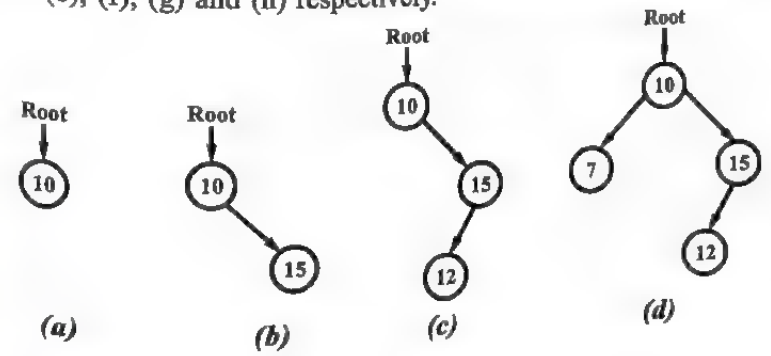
First of all, we initialize the tree, to create an empty tree we must initialize root to null. The first node will be inserted into the tree as shown in fig. 5.3 (a).

Since 15 is greater than 10, it must be inserted as the right child of the root as shown in fig. 5.3 (b).

Again 12 is larger than root, it must go to the right subtree of the root. Now it is smaller than 15, so it must be inserted as the left child of the root as shown in fig. 5.3 (c).

Next number 7 is smaller than the root, thus it must be inserted as the left child of the root as shown in fig. 5.3 (d).

Similary 8, 18, 6 and 20 can be inserted at the proper place as shown in fig. 5.3 (e), (f), (g) and (h) respectively.



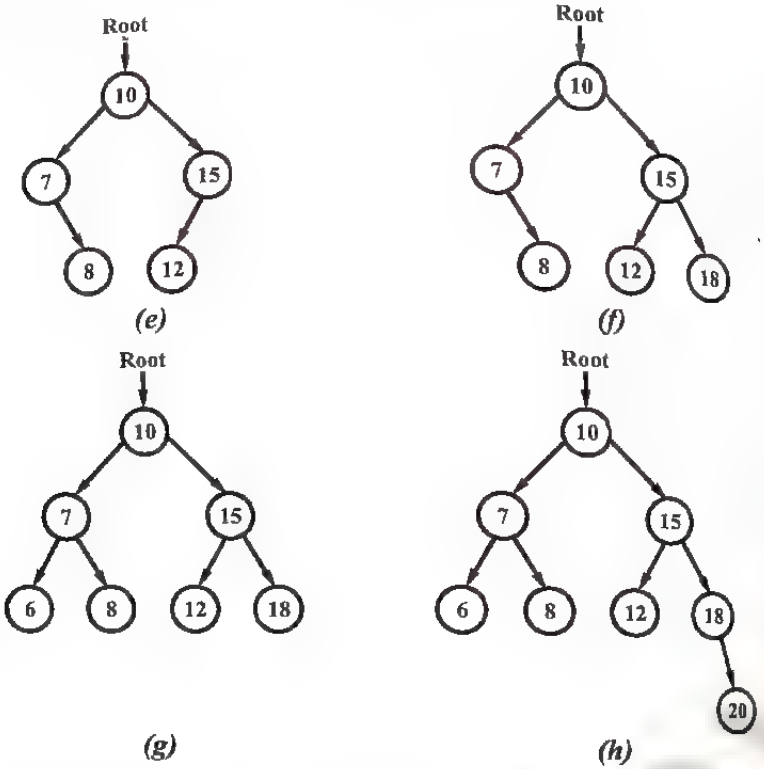


Fig. 5.3

To insert a new value V into a binary search tree T, we use the procedure 5.2 TREE-INSERT. The procedure is passed a node z for which key [z] = v, left [z] = NIL, and right [z] = NIL. It then modifies T and some of the fields of z in such a way that z is inserted into an appropriate i.e., correct position in the tree.

Algorithm 5.2

```
TREE-INSERT (T, z)
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.      do y ← x
5.          if key[z] < key[x]
6.              then x ← left[x]
7.              else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.     then root[T] ← z
11.     else if key[z] < key[y]
12.         then left[y] ← z
13.         else right[y] ← z
```


TREE-INSERT procedure begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the while loop in lines 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of key[z] with key[x], until x is set to NIL. Then this NIL will occupy the position where we wish to place the input item z. Lines 8-13 set the pointers that cause z to be inserted. The time complexity of this procedure runs O(h) time on a tree of height h.

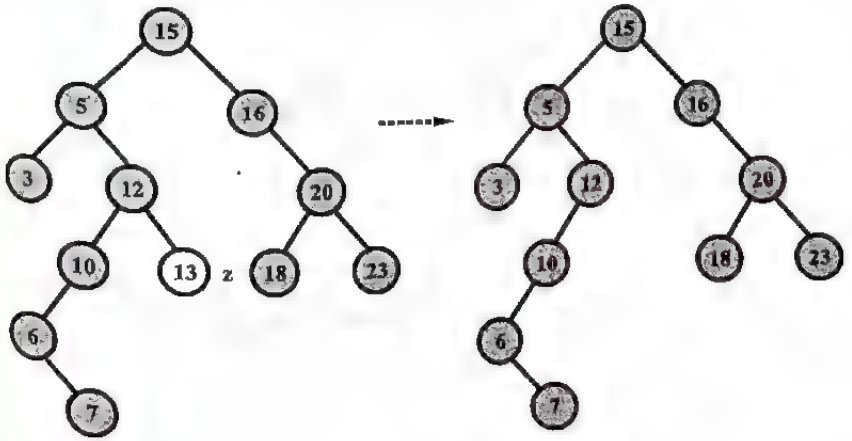
Q.5. Explain with the help of algorithm, to delete an item from the binary search tree. Also give an example to delete an item from the binary search tree.

Or

Explain the function for deletion of a node from binary search tree.

(R.G.P.V., Dec. 2015)

Ans. Another important function for maintaining binary search tree is to delete a specific node from the tree. The method to delete a node or item depends on the specific position of the node in the tree. The procedure for deleting a given node z from a binary search tree takes as an argument a pointer z. The procedure takes the three cases as shown in fig. 5.4. If z has no children, we modify its parent p[z] to replace z with NIL as its child. If the node has only a single child, we "split out" z by making a new link between its child and its parent. Finally, if the node has two children, then we splice out z's successor y, which has no left child and replace the contents of z with the contents of y. The three cases are described below in the fig. 5.4. Fig. 5.4 deletes a node z from a binary search tree. In each case, the node actually removed is lightly shaded. (a) If z has no children, we just remove it (b) If z has only one child, we splice out z (c) If z has two children, we splice out its successor y, which has at most one child and then replace the contents of z with the contents of y.



(a)

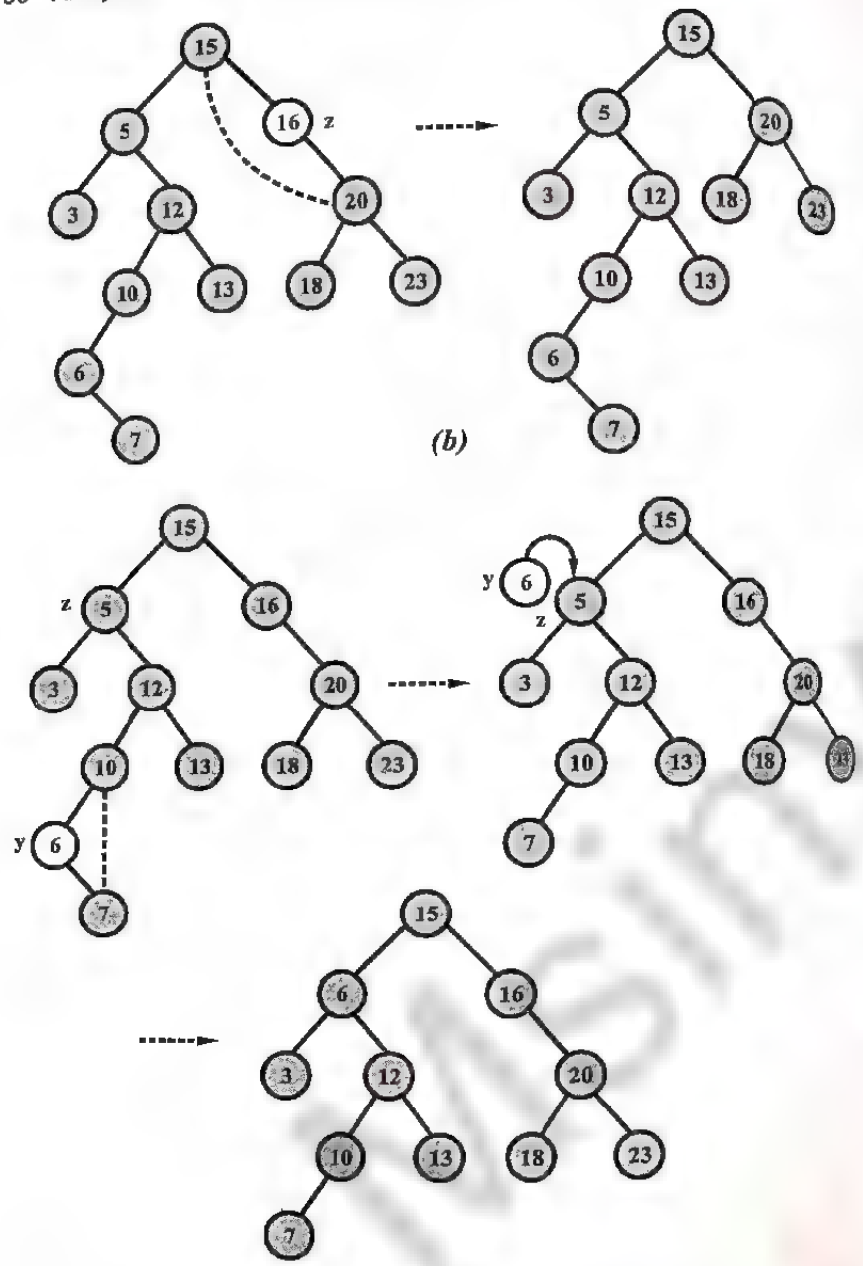


Fig. 5.4 Deleting an Item in Binary Search Tree

Algorithm 5.3

TREE-DELETE (T, z)

```
1.  if left[z] = NIL  or right[z] = NIL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR (z)
4.  if left[y] ≠ NIL
5.       then x ← left[y]
6.       else x ← right[y]
7.  if  x ≠ NIL
8.       then p[x] ← p[y]
9.  if p[y] = NIL
10.      then root[T] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
13.           else right[p[y]] ← x
14. if y ≠ z
15.      then key[z] ← key[y]
16.           ▷ if y has other fields, copy them, too.
17. return y
```


In lines 1-3, the algorithm 5.3 determines a node y to splice out. The node y may be the input node z (if z has atmost 1 child) or the successor of z (if z has two children). Then, in lines 4-6, x is set to the non-NIL child of y, or to NIL if y has no children. The node y is spliced out in lines 7-13 by modifying pointers in p[y] and x. Splicing out y is however complicated by the need for proper handling of the boundary conditions, which occur when x = NIL or when y is the root. Finally, in lines 14-16, if the successor of z was the node spliced out, the contents of z are moved from y to z, overwriting the previous contents. The node y is returned in line 17 so that the calling procedure can recycle it via the free list. The procedure runs in O(h) time on a tree of height h.

Q.6. Define height balanced trees. Write the properties of it. ***(R.G.P.V., Dec. 2011)***

Or

What is AVL tree ? Discuss its properties. ***(R.G.P.V., June 2014)***

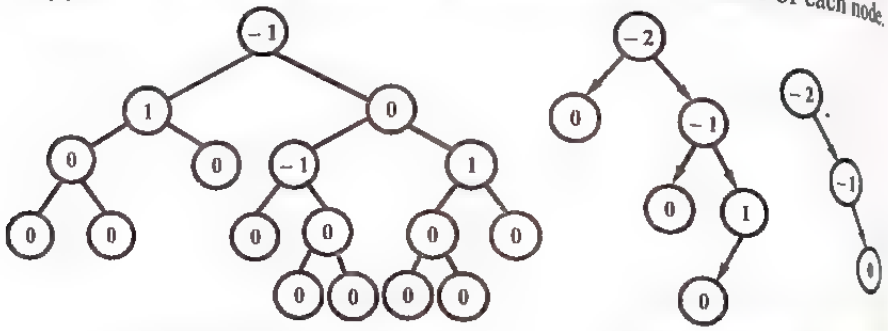
Or

What is height balanced trees ? Why to be balanced a height in a trees? ***(R.G.P.V., May 2018)***

Ans. A ***height balanced tree*** is a binary tree in which the difference in heights between the left and the right subtree is not more than one for every node. The height of a binary tree is defined as the maximum level of its leaves. The height of a null tree is defined as – 1. This property was first described in 1962 by two Russian Mathematicians, G.M. Adel'son – Vel'skii and E.M. Landis. Hence, the resulting binary trees are also called **AVL** ***trees*** in their honor.

To maintain, the height balanced property, it is actually not necessary to know the height of each subtree. We can obtain it by maintaining at each node a ***balance*** factor that indicates the difference in heights of the left and right

subtrees. Each node in a balanced binary tree has a balance of 1, – 1 or 0 depending on whether the height of its left subtree is greater than, less than or equal to the height of its right subtree. A value of 1 shows that the left child is 'heavier' and there is a path from root to leaf in the left child subtree of length n, whereas the longest path in the right child subtree is length n – 1. A balance factor of 0 will show that the longest paths in the two child subtrees are equal while a value of – 1 shows that the right child possesses the longest path. Fig. 5.5 (a) shows a balanced binary tree. It also shows the balance of each node.



(a) Balanced Binary Tree *(b) Unbalanced Binary Tree*

Fig. 5.5

In a balanced tree each node must be in one of the three states – 1, 0, 1. If there is a node in a tree where this is not true, then such a tree is said to be ***unbalanced.*** For example the trees shown in fig. 5.5 (b) are unbalanced trees.

Q.7. Write an algorithm for inserting a new node into an AVL tree.

Ans. For inserting a new node into the AVL tree, an algorithm is given in algorithm 5.4.

Here, 'newnode' indicates that new node to be inserted into the tree.

bf denotes balance factor, type of balance factor is (LH, FQ, RH)

LH – denotes left higher

EQ – denotes equal height

RH – denotes right higher.

AVL tree is pointed by root node, first of all we check whether root node is nil or not. If root node is nil, then it indicates that tree is empty and newnode is to be inserted at the position of root of AVL tree. Its left and right subtrees are set to nil.

Secondly, if root node is not nil, then, we check the key value of root node to key of newnode if both are equal then, as duplication is not allowed, entry of new node is restricted. If key value does not match with key of new node then newnode is inserted either in left subtree of root or right subtree of root depending on its key value.

If newnode key is less than key of root node then we call algorithm insert AVL(left, newnode, tallersubtree) otherwise we call algorithm insert AVL(right, newnode, tallersubtree).

Here tallersubtree is a boolean type. It returns true if after insertion height of a subtree is increased else, it returns false.

Algorithm 5.4 Insertion into an AVL Tree

```
1.  Algorithm InsertAVL(var root: treepointer;
                newnode: treepointer; var taller: Boolean);
    {Pre:  The root of the AVL tree is pointed to by root, and newnode is a
           new node to be inserted into the tree.
    Post:  Newnode has been inserted into the AVL tree with taller equal to
           true if the height of the tree has increased, false otherwise.
    Uses : InsertAVL recursively, RightBalance, LeftBalance.}
2.  var
3.  tallersubtree: Boolean;   {Has the height of a subtree increased?}
4.  begin
5.      if root = nil then begin
6.          root := newnode;
7.          root↑.left := nil;
8.          root↑.right := nil;
9.          root↑.bf := EQ;
10.         taller := true
11.     end
12.     else with root↑ do
13.         if newnode↑.entry.key = entry.key then
14.             Error('Duplicate key is not allowed in AVL tree.')
15.         else if newnode↑.entry.key < entry.key then begin
                                              {Insert in the left subtree}
16.             InsertAVL(left, newnode, tallersubtree);
17.             if tallersubtree then          {Change the balance factors.}
18.                 case bf of
19.                     LH: LeftBalance; -
20.                     EQ: begin bf := LH; taller := true end;
21.                     RH: begin bf := EQ; taller := false end
22.                 end
23.             else taller := false
24.         end
25.         else begin                         {Insert in the right subtree}
```

```
            InsertAVL(right, newnode, tallersubtree);
            if tallersubtree then
                case bf of
                    LH: begin bf := EQ; taller := false end;
                    EQ: begin bf := RH; taller := true end;
                    RH: RightBalance
                end
            else taller := false
        end
    end;                                        {procedure insertAVL}
```

As we must keep track of whether an insertion has increased the height or not. So that balance factor can be changed appropriately.

Q.8. How can rotation be done in AVL trees ? Explain.

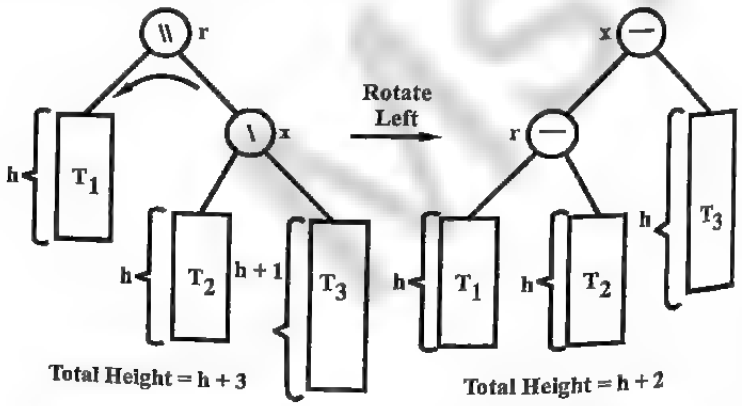
Ans. In the cases when insertion of a newnode into the AVL tree result in increase of height of longer subtree that result into a tree that has difference of height of 2 between left and right subtree of root. Now, resulting tree does not satisfy the requirements of an AVL tree.

For restoring the balance, we have to rebuild part of the tree –

For example, suppose that new node is inserted into the right subtree, its height has increased, and the original tree was right higher. Now we wish to balance this tree.

Suppose, root node of AVL tree is r and its right child is denoted by x.

(i) Right Higher Left Rotation – Right subtree has height 2 more than left subtree as shown in fig. 5.6 (a). The action needed in this case is "***left rotation***". In this case as shown in fig. 5.6 (b), we rotate the tree leftward,



(a) Before Rotation ***(b) After Rotation***

Fig. 5.6 Restoring Balance by a Left Rotation

node x becomes the new root of the tree and node r i.e., root node becomes the left child of new root x.

After rotation the height of rotated tree is decreased by 1.

Algorithm for left rotation is given in algorithm 5.5.

Algorithm 5.5 Left Rotation Algorithm

```
1.  Algorithm RotateLeft(var p: treepointer);
    {Pre:  p is the root of the nonempty AVL subtree being rotated, and its
           right child is nonempty.
    Post:  The right child of p becomes the new p. The old p becomes the
           left child of the new p.}
2.  var rightchild: treepointer;
3.  begin                                   {procedure RotateLeft}
4.      if p = nil then
5.          Error('It is impossible to rotate empty tree in RotateLeft.')
6.      else if p↑.right = nil then
7.          Error('It is impossible to make empty subtree the root in
            RotateLeft.')
8.      else begin
9.          rightchild := p↑.right;
10.         p↑.right := rightchild↑.left;   {Move right subtree to intermediate
            node.}
11.         rightchild↑.left := p;          {Drop p into left subtree}
12.         p := rightchild                 {Change p to former right subtree.}
13.     end
14. end;                                    {procedure RotateLeft}
```

In above algorithm first of all we check that the root of AVL tree is nil or not. If it is nil, indicates that tree is empty and so no rotation is possible, and an error message is given.

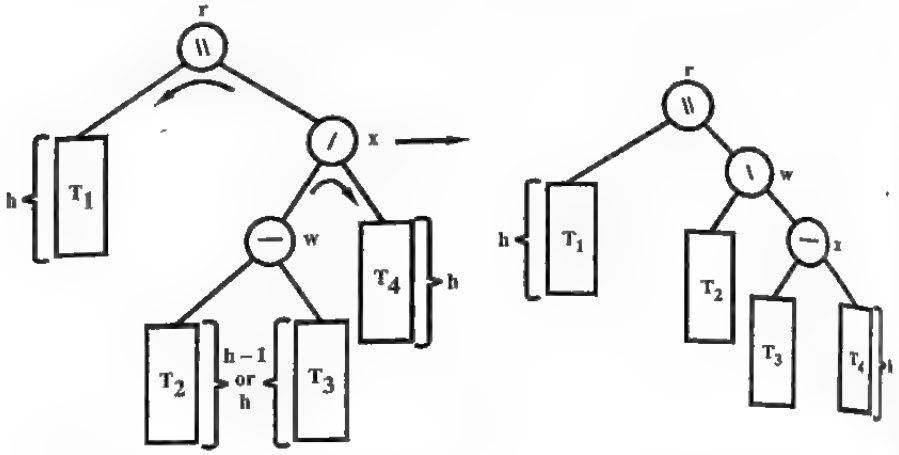
Secondly, we check that whether the right child of root is nil or not. If it is nil then left rotation is not allowed, as an empty node cannot be root of an AVL tree so again an error message is given.

Finally if rightchild of root is not empty then left rotation is applied.

Here, consider that p denotes the root node. Node p is moved leftward by using an intermediate node called rightchild.

(ii) Left Higher – In the case, when the height of left subtree is higher than the height of right subtree. The action needed in this case is ***right rotation***.

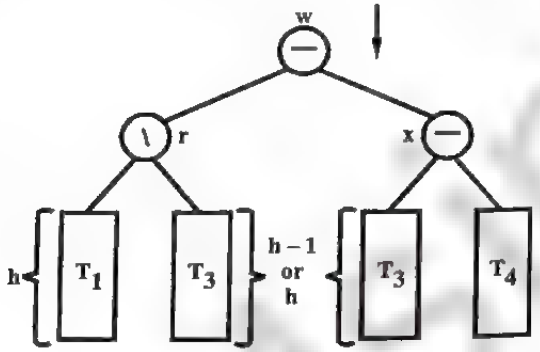
For example, see fig. 5.7 (a). Here, height of left and right subtrees of node r differ by 2. So, we have to first right rotate tree rooted at node x.



(a) One of T_2 or T_3 has Height h

Total Height = h + 3

(b) Height h + 3



(c) Height h + 2

Fig. 5.7 Restoring Balance by a Double Rotation

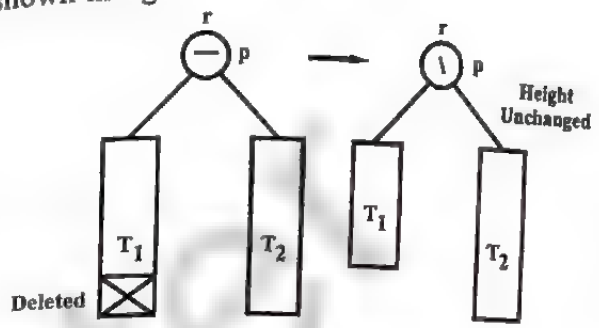
Q.9. Define height balanced tree. Explain all the rotations perform to balance the tree with example. ***(R.G.P.V., June 2010)***

Ans. Refer to Q.6 and Q.8.

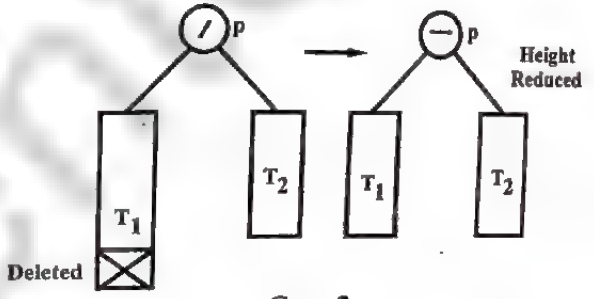
Q.10. How deletion can be performed on an AVL tree ?

Ans. When we delete a node from an AVL tree, if its height is unbalanced then we need to rotate it either left or right so as to keep its height balanced or as an AVL tree.

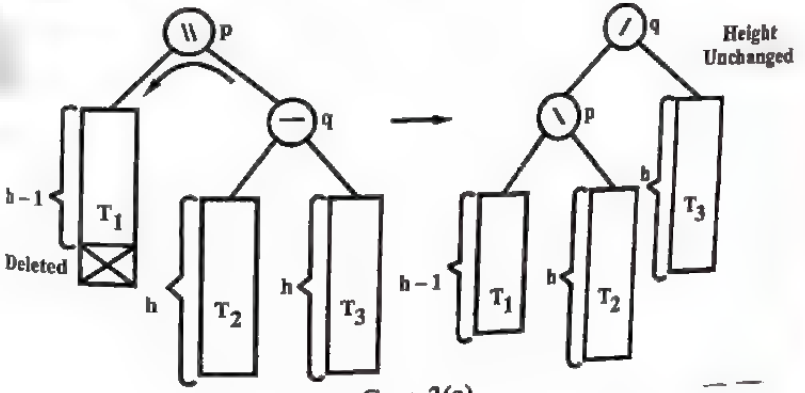
This is shown in fig. 5.8.



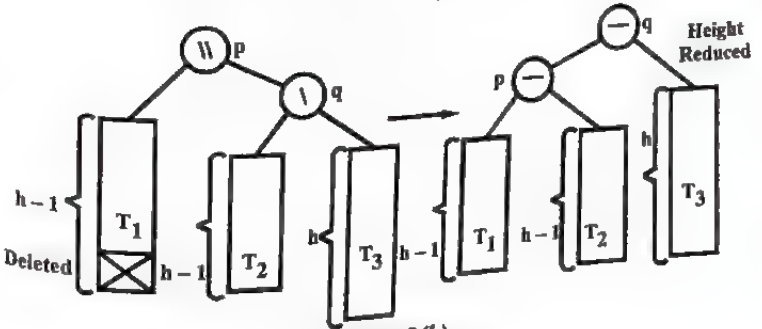
Case 1



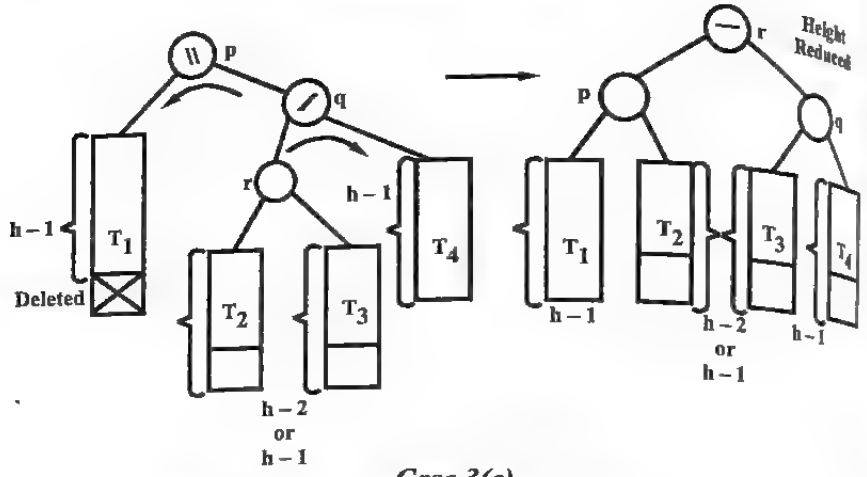
Case 2



Case 3(a)



Case 3(b)



Case 3(c)

Fig. 5.8 Sample Cases, Deletion from an AVL Tree

Case 1 – In first case after deletion of a node from left subtree of root node, the height of tree remain unchanged. The resulting tree is an AVL tree, so no need of rotation.

Case 2 – In this case after deleting a node from left subtree of root node height of the tree is reduced, but now the resulting tree is height balanced. It has equal height in both right and left subtrees.

Case 3 – In this case height of left subtree of root node is h while the height of right subtree of root node is h + 1 i.e., difference of 1, we say it right higher.

If we delete a node from left subtree than it results into a tree that has height difference of 2.

For balancing it we rotate it left. By this resulting tree has height unchanged.

Case 4 – In this case height of left-subtree of root node of tree is h while the height of right subtree is h + 1.

If we delete a node from left subtree of root node then the difference between the heights of tree is 2.

For balancing it we have to rotate left corresponding to root node p. The resulting tree is AVL tree with height reduced by 1.

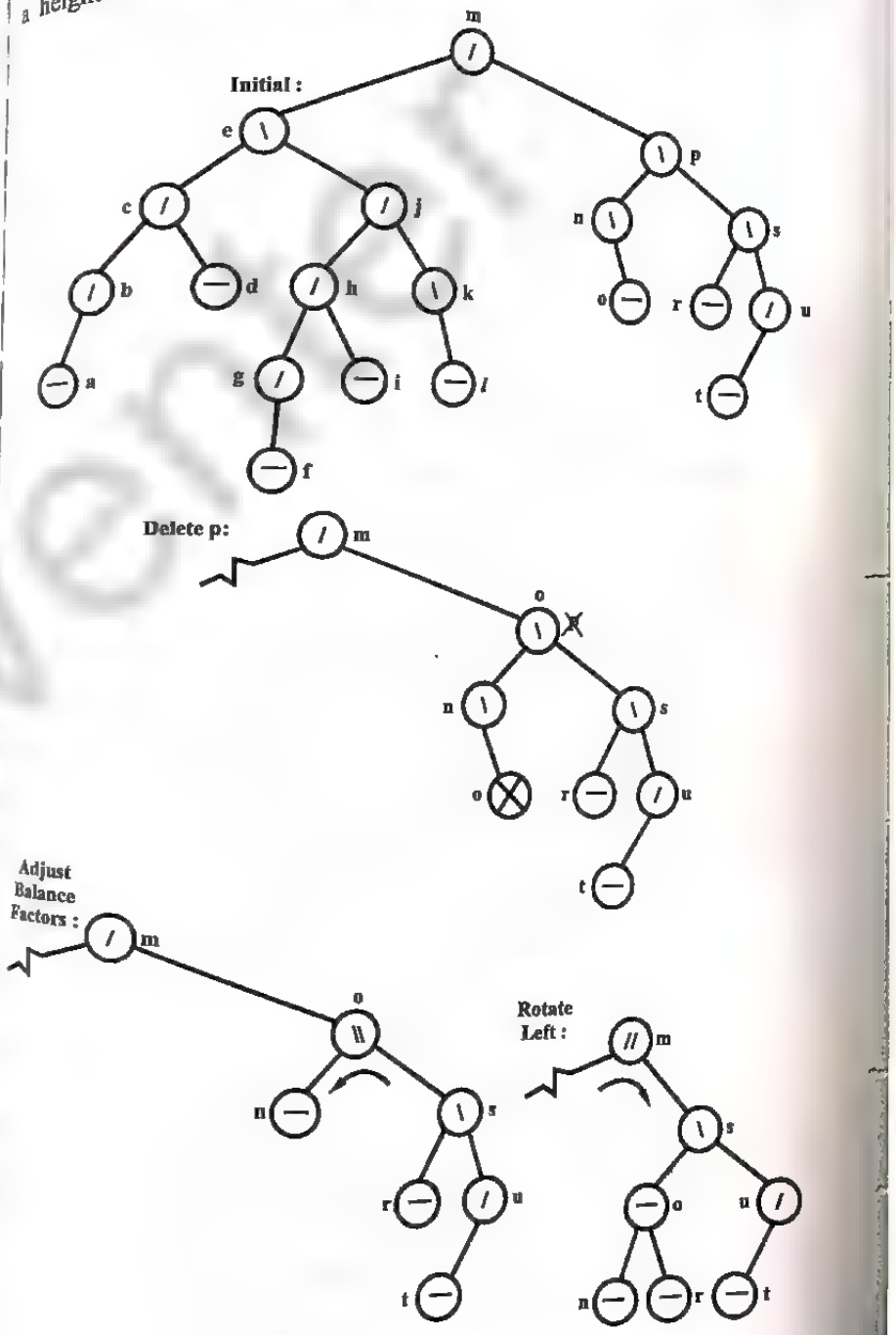
Case 5 – In this case left subtree has height h while right subtree has height either h + 1 or h. i.e., a difference by a factor of 1 or 0 respectively.

If a node is deleted from left subtree than the difference between their height is 2.

For balancing the tree two rotations are required –

(i) Right rotation corresponding to node p.

(ii) Left rotate the tree corresponding to root node p. It results into a height balanced tree rooted at node r.



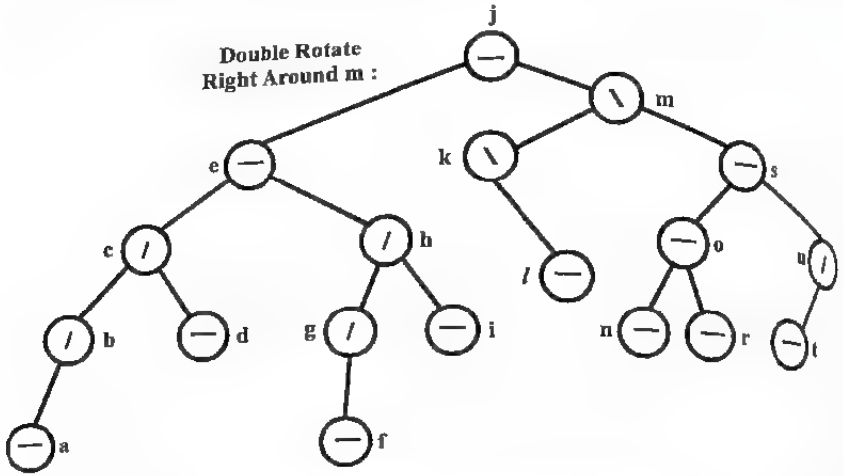


Fig. 5.9 Deletion from an AVL Tree

Q.11. Explain the concept of height balanced trees with their operations. ***(R.G.P.V., Dec. 2016)***

Ans. Refer to Q.6, Q.7, Q.8 and Q.10.

Q.12. Explain 2-3 trees with examples. ***(R.G.P.V., Dec. 2013)***

Or

Explain 2-3 trees with the help of suitable example.

(R.G.P.V., June 2014, Dec. 2014, 2015)

Or

Write a short note on 2-3 tree. ***(R.G.P.V., Dec. 2010)***

Ans. A 2-3 (or 3-2) tree is a tree in which each vertex, except leaf, has 2 or 3 sons and one or two keys per node, and every path from the root to leaf is of the same length.

The tree consisting of a single vertex is a 2-3 tree. Fig. 5.10 shows the two 2-3 trees with six leaves.

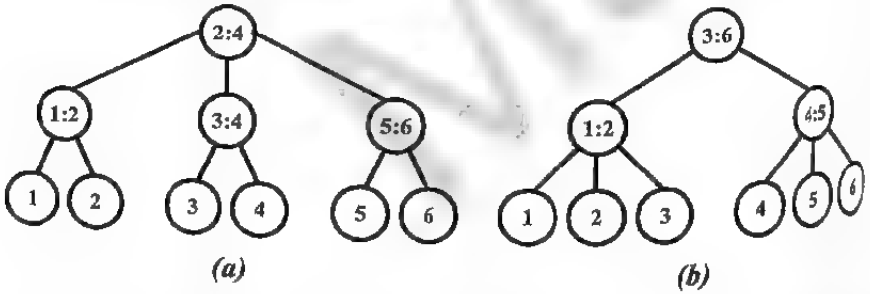


Fig. 5.10 2-3 Trees

Let T be a 2-3 tree of height h. The number of vertices of T is between $(2^{h+1}-1)$ and $(3^{h+1}-1)/2$, and the number of leaves is between 2^h and 3^h.

Q.13. Write short note on B-tree. ***(R.G.P.V., May 2018, Nov. 2018)***

Ans. B-tree of order M is an M-way tree in which

(i) All leaves are on the same level.

(ii) All internal nodes except the root have atmost M (nonempty) children and at least (M/2) (nonempty) children.

(iii) The number of keys in each internal node is one less than number of its children in the fashion of a search tree.

(iv) The root has atmost M children, but may have as few as 2 if it is not a left, or none, if the tree consists of the root alone.

For example, fig. 5.11 shows a B-tree of order 5, as here all leaves are on same level and it also satisfies all other conditions of B-tree.

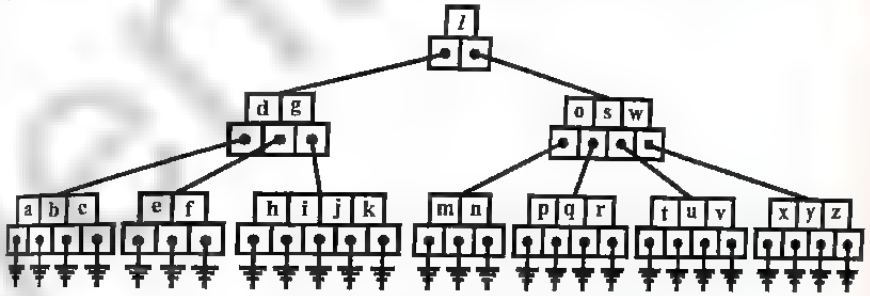


Fig. 5.11 A B-tree of Order 5

While fig. 5.12 does not show a B-tree.

As in fig. 5.12 some nodes have empty children, and the leaves are not all on the same level. While fig. 5.11 is a B-tree of order 5 whose keys are the 26 letters of the alphabet.

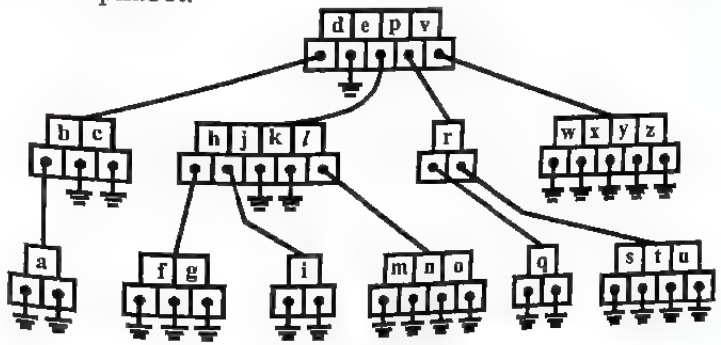


Fig. 5.12 A 5-way Search Tree

In 1970, J.E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4 trees, in which every internal node has either two or three children. B-trees were introduced by Bayer and McCreight in 1972 [18]; they did not explain their choice of name.

Q.14. What are 2-3 trees and B-trees ? Explain with example. ***(R.G.P.V., June 2011)***

Ans. Refer to Q.12 and Q.13.

Q.15. Define B-trees. Prove that if $n \geq 1$, then for any n-key, B tree of height h and minimum degree $t \geq 2$, $h \leq \log_t \{(n+1)/2\}$. (R.G.P.V., Dec. 2007)

Ans. **B-trees** – Refer to Q.13.

Proof – If a B-tree has height h, the number of its nodes is minimized when the root contains one key and all other nodes contain $t-1$ keys. In this case, there are 2 nodes at depth 1, 2t nodes at depth 2, $2t^2$ nodes at depth 3, and so on, until at depth h there are $2t^{h-1}$ nodes. Fig. 5.13 shows such a tree for h = 3. Thus, the number n of keys satisfies the inequality.

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

which implies the theorem.

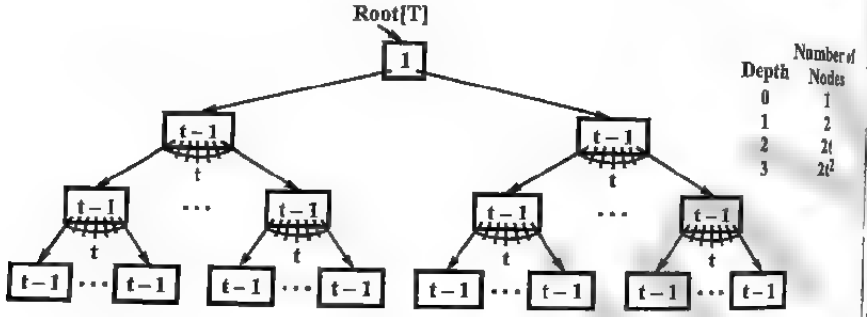


Fig. 5.13 A B-tree of Height 3 Containing a Minimum Possible Number of Keys

Q.16. What is a B-tree ? Write down the properties of a B-tree. Illustrate your answer with an example. ***(R.G.P.V., June 2017)***

Ans. Refer to Q.13.

Order – Order of B-tree defined as the minimum number of keys in a non-root node (i.e., $(n-1)/2$).

Degree – Degree of B-tree is the maximum number of sons (i.e., n)

Example – Refer to Q.13.

Q.17. What are B-trees ? Write down its properties. What is the need for B-tree ? What is the height of a B tree of order m ? (R.G.P.V., May 2019)

Ans. Refer to Q.13 and Q.16.

A B-tree can be used as an index file. The actual data records are stored elsewhere on disk. When one looks up and finds a target key, what one finds is the target key and an associated pointer (disk block number of whatever) to the appropriate data record. This means that one data file could have several such indices, each giving an ordering by a different key field, something highly desirable to have.

Height Analysis – Consider a B-tree (of order m) of height h. Since every node has degree at most m, then the tree has at most m^h leaves. Since each node contains at least one key, this means that $n \geq m^h$. We assume that m is a constant, so the height of the tree is given asymptotically by

$$h \leq \log_m n = \frac{\lg n}{\lg m} \in O(\log n)$$

This is very small as m gets large. Even in the worst case (excluding the root) the tree essentially is splitting m/2 ways, implying that

$$h \leq \log_{(m/2)} n = \frac{\lg n}{\lg (m/2)} \in O(\log n)$$

Thus, the height of the tree is O(log n) in either case. For example, if m = 256 we can store 100,000 records with a height of only three. This is important, since disk accesses are typically many orders of magnitude slower than main memory accesses, it is important to minimize the number of accesses.

Q.18. How multiway search is different from binary search tree ? ***(R.G.P.V., Dec. 2016)***

Ans. Difference between multiway search tree and binary search tree are as follows –

S.No.	*M-way Search Tree*	*Binary Search Tree*
(i)	It is a tree data structure with each of its node can hold at most M branches.	It is a tree data structure with each of its node can hold at most 2 branches.
(ii)	A node in M-way search tree can store more than one keys (M – 1 keys).	A node of binary search tree can store only one key.
(iii)	If a node in M-way search tree is having k keys, where k < M, then that node can have atmost k + 1 branches.	Every node is binary search tree, if not empty then can have atmost two branches.

Q.19. How insertion can be done into a B-tree ? Explain.

Ans. Insertion of a node into a B-tree grows at the root as it has a characteristic that all leaves be on the same level. While binary search tree grows at their leaves.

If each time insertion of a node is done into an empty node then we just insert the new node and the process finishes. If we insert newnode into a full node then this node splits in between from median.

This median goes in higher level and becomes new root of two children that generated above. Both of these nodes are on same level.

When a search is later made through the tree, therefore a comparison with the median key will serve to direct the search into the proper subtree. For example suppose we have following set of keys to be inserted into a B-tree -

a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p.

We are inserting above set into a B-tree of order 5.

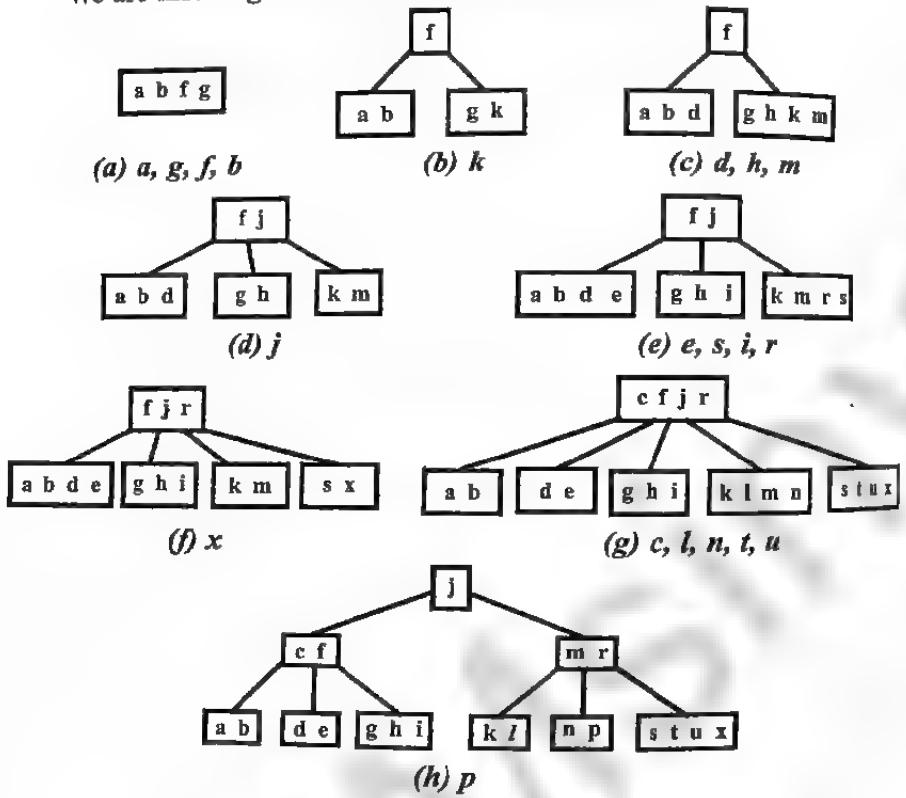


Fig. 5.14 Growth of a B-tree or Insertion into a B-tree

Initially B-tree is empty. Insertion can be shown by fig. 5.14.

(i) At each time when we insert a new node into an empty node then newnode is sorted with already present keys as shown by fig. 5.14 (a).

(ii) When a new key k is added after adding 4 keys a, b, f, g as B-tree is of order 5 so above node is full, and on insertion of node k it splits into two nodes one having keys a, b and other one having keys g, k and breaks at median of which go in higher level become new root of the tree as shown in 5.14 (b).

(iii) On adding d it gets added to a, b resulting as a, b, d. On adding h and m, they get added with g and k resulting g, h, k, m as shown in fig. 5.14 (c).

(iv) Further when node j is added, it splits from median i.e., j and this median gets attached to i, resulting in fig. 5.14 (d).

(v) Further additions of e, s, i, r do not result in any splitting shown in fig. 5.14 (e).

(vi) On addition x again a split takes place at median r, shown in fig. 5.14 (f).

(vii) Further c, j, n, t, u are added without any more splitting shown in fig. 5.14 (g).

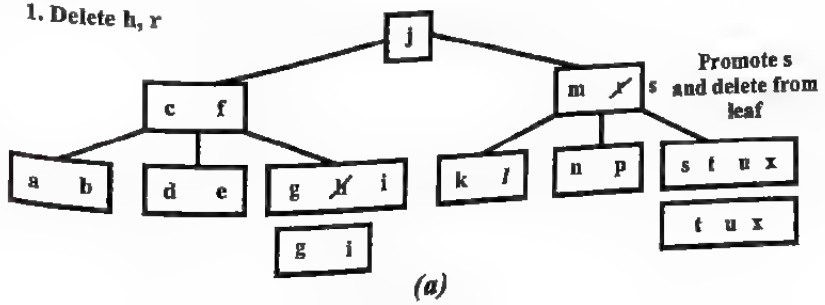
(viii) When node p is added , again a split takes place at median m but now upper level is also full, so further breakage takes place at median j, resulting in j at one more higher level becomes new root of tree as shown in fig. 5.14 (h).

Q.20. How can deletion be done from a B-tree ? Explain.

Ans. For deleting an entry from a B-tree, we should always check whether the node from which an entry is to be deleted has minimum number of entries or not.

If it has minimum number of entries then deletion is complete, otherwise we first look at the two leaves (or, in the case of a node on the outside, one leaf) that are immediately adjacent to each other and are children of the same node. If one of these has more than the minimum number of entries, then one of them can be moved into the parent node, and the entry from the parent moved into the leaf where the deletion is occurring.

If finally, the adjacent leaf has only the minimum number of entries, then the two leaves and the maximum number of entries allowed. If this step leaves the parent node with too few entries, then the process propagates upwards. In the limiting case, the last entry is removed from the root, and then the height of the tree decreases. For example, the process of deletion in a B-tree is shown in fig. 5.15 (a).



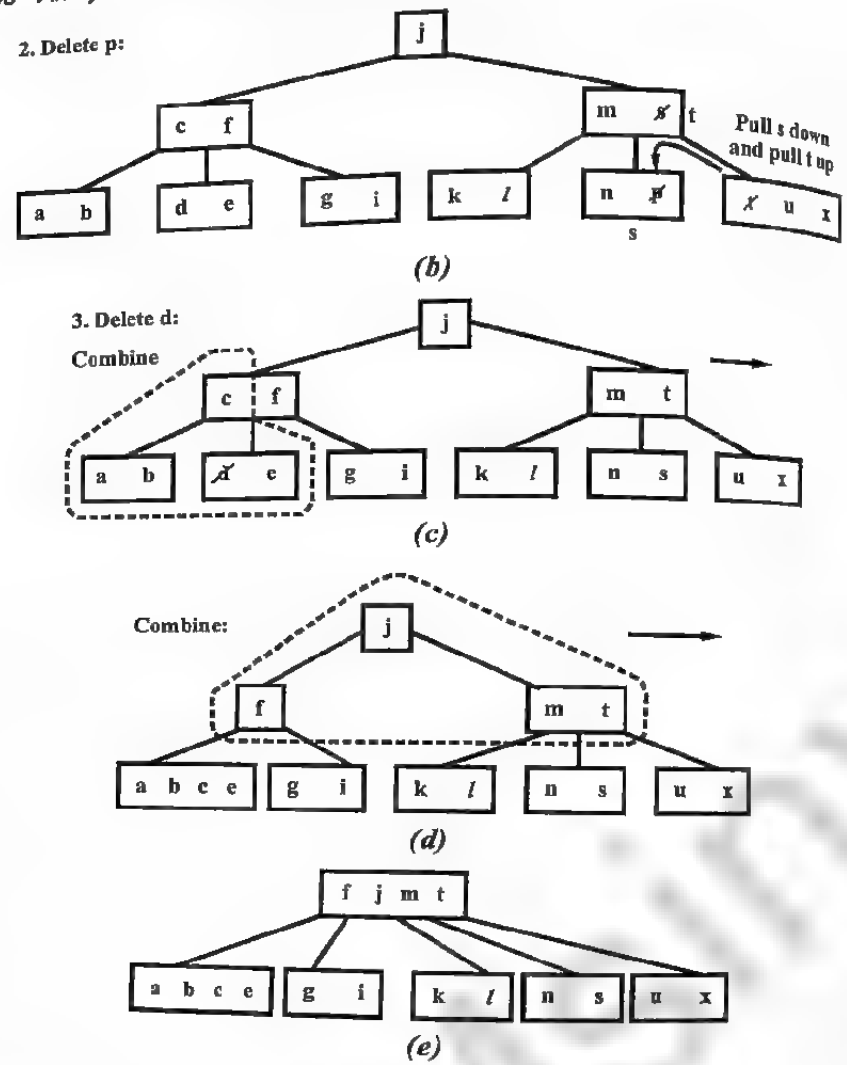


Fig. 5.15 Deletion from a B-tree

(i) h – If we delete entry h from above B-tree of order 5, then the resulting node after deletion has 2 entry i.e., minimum number of entries required. So, here deletion gets finished and no further action is required as shown in fig. 5.15 (a).

(ii) r – If we delete entry r from the B-tree, then after deletion the resulting node has only one entry, i.e., less than minimum entries allowed, so we look for its successor (s) or predecessor (p).

Node having p has only two entries, so here removal is not allowed. We remove successor s and replace r as shown in fig. 5.15 (a).

(iii) p – For deleting p, the node after its deletion has only one entry so we look for its successor s as its predecessor in n.

If we replace p with s then node having s, has only one entry so again we look for successor of S, i.e., t resulting into a tree as shown in fig. 5.15 (b).

(iv) d – For deletion d, we have to replace it by c, but resulting node has only one entry, i.e., f. So again we look for predecessor of c, i.e., a, b.

Now, a deletion of b is also not possible, for this we combine two nodes and delete entry d resulting in a tree shown in fig. 5.15 (d). But here also nodes have only one entry f, i.e., less than minimum possible number. Again we combine three nodes resulting into a tree as shown in fig. 5.15 (e).

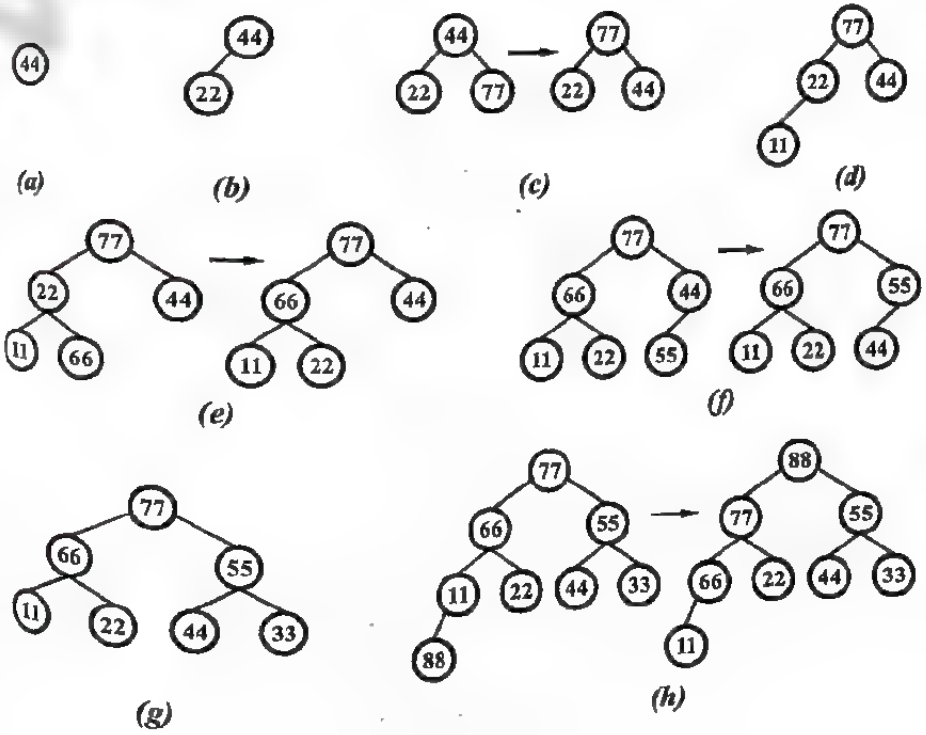
NUMERICAL PROBLEMS

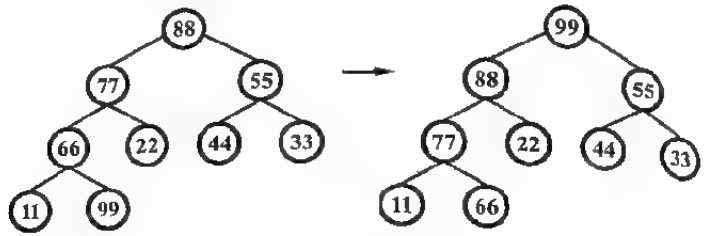
Prob.1. Compare heap and binary search tree. Construct a heap and a binary search tree of the following data –

44, 22, 77, 11, 66, 55, 33, 88, 99. ***(R.G.P.V., Dec. 2013)***

Sol. **Comparison between Binary Search Tree and Heap** – Refer to Q.2.

Construction of Heap – The construction of max-heap is shown in fig. 5.16.





(i)

Fig. 5.16

Creation of Binary Search Tree – The creation of binary search tree is as follows –

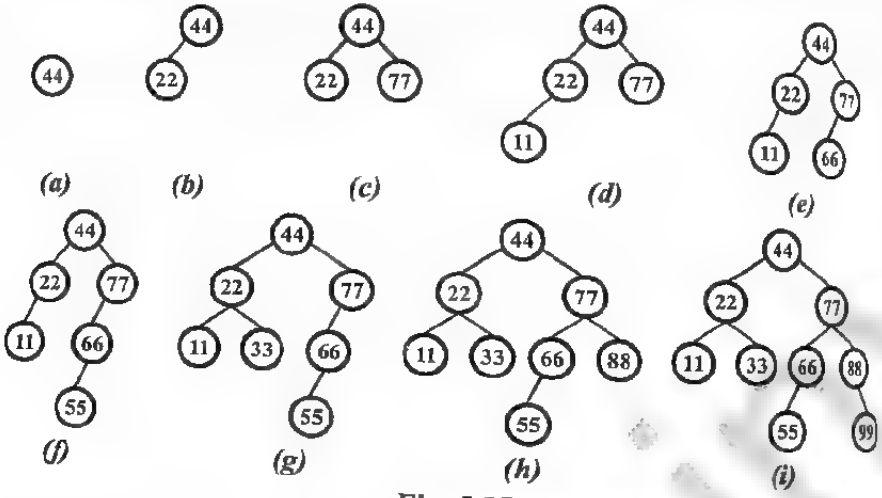


Fig. 5.17

Prob.2. The post order traversal of a binary tree T is DFEBGLJKHCA and inorder traversal of T is DBFEAGCLJHK. Construct the binary tree T and also writes the steps to construct the binary tree in postorder-inorder combination.

(R.G.P.V., June 2016)

Sol. The steps to construct the binary tree in postorder-inorder combination are as follows –

(i) Postorder – D F E B G L J K H C A
Inorder – D B F E A G C L J H K
Left subtree Right subtree

Since root of a tree is visited at the last in postorder traversal. Hence, A is the root of binary tree. Observing the inorder traversal, we see that the nodes before A make the left subtree and nodes after A make right subtree of a binary tree.

(ii) The root of left subtree is found by the last node in the postorder traversal of left subtree. That is B; D is left child of B by observing the inorder traversal and the right subtree of B consists of F and E. Again E is the root of this subtree in postorder traversal and F is the left child of E by observing the inorder traversal.

(iii) The root of right subtree is found by the last node in the postorder traversal of right subtree. That is C; G is the left child of C by observing the inorder traversal and the right subtree of C consists of L, J, H and K. Again H is the root of this subtree in postorder traversal and K is the right child of H by observing the inorder traversal and left subtree of H consists of L and J. Again J is the root of this subtree in postorder traversal and L is the left child of J by observing the inorder traversal.

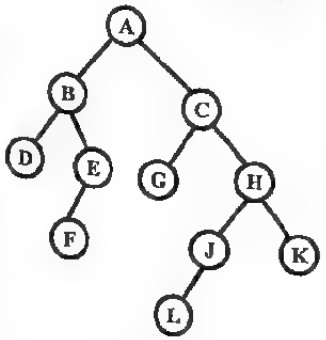


Fig. 5.18

The constructed binary tree is shown in fig. 5.18.

Prob.3. A binary tree T has 9 nodes the inorder and preorder traversal of T yield the following sequence of nodes –

Inorder – E A C K F H D B G
Preorder – F A E K C D H G B
Draw the tree T.

(R.G.P.V., Dec. 2017)

Sol. The steps to construct the binary tree in preorder and inorder combination are as follows –

(i) Preorder – F A E K C D H G B
Inorder – E A C K (Left) F (Root node) H D B G (Right)

Since, in preorder traversal the first node is the root node. Hence F is the root node of the given binary tree.

Now from the given inorder traversal we conclude that nodes before F make a left subtree and the nodes after F make the right subtree.

(ii) On considering the left subtree E A C K, the root will be 'A' as A occurs first in preorder traversal as compared to E, C and K nodes. Similarly as compared to the above process, E makes a left subtree and the nodes after A make the right subtree.

(iii) On comparing the right sub-tree, the root node will be 'D' as it occurs first in preorder traversal as compared to H, B and G. Hence the nodes before D make the left subtree and the nodes after D make the right subtree.

Hence, following the same process we obtain the binary tree as shown in fig. 5.19.

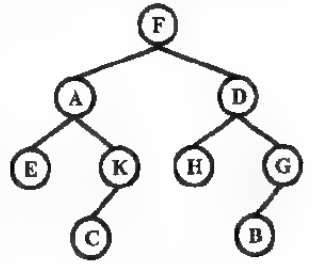
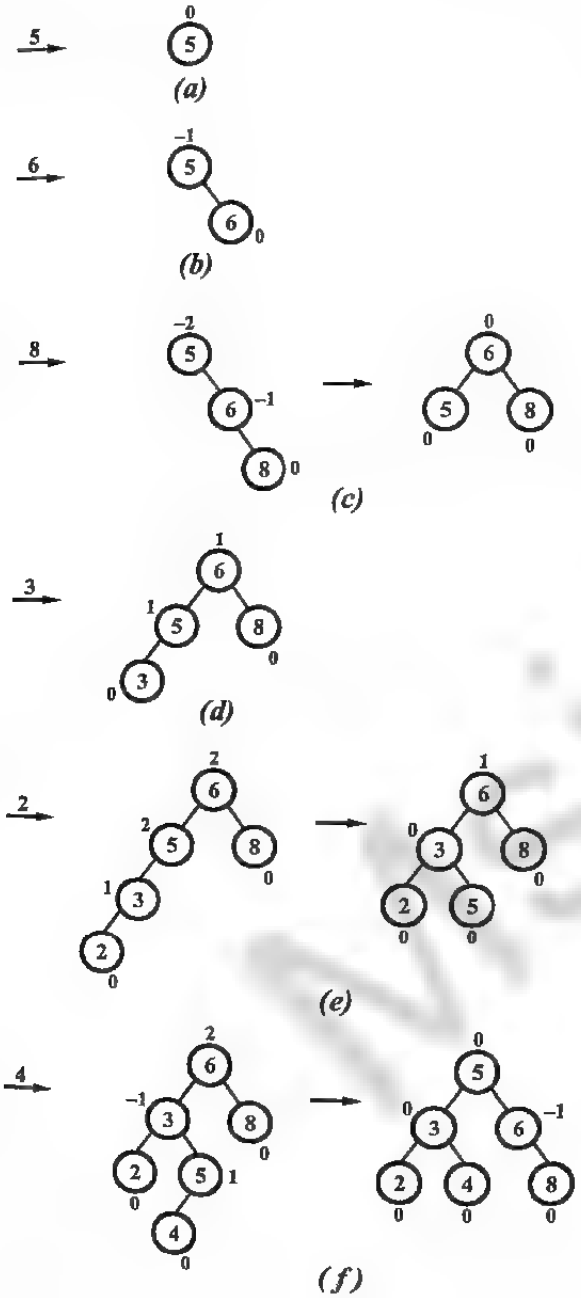


Fig. 5.19

Prob.4. Construct an AVL tree for the following list {5, 6, 8, 3, 2, 4, 7} by inserting the elements successively, starting with empty tree. ***(R.G.P.V., June 2008, Dec. 2014)***

Sol. The construction of AVL tree is as follows –



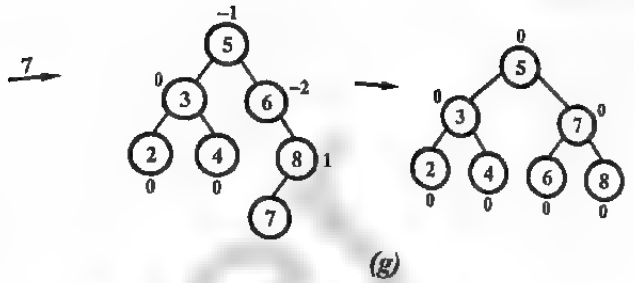


Fig. 5.20

Prob.5. Compare AVL trees with binary search trees. Insert the following keys into an AVL tree –

28, 32, 12, 5, 84, 53, 91, 35, 3, 11 ***(R.G.P.V., Dec. 2011)***

Sol. **Comparison** – Refer to Q.1 and Q.6.

Fig. 5.21 shows the insertion of given keys into an AVL tree.

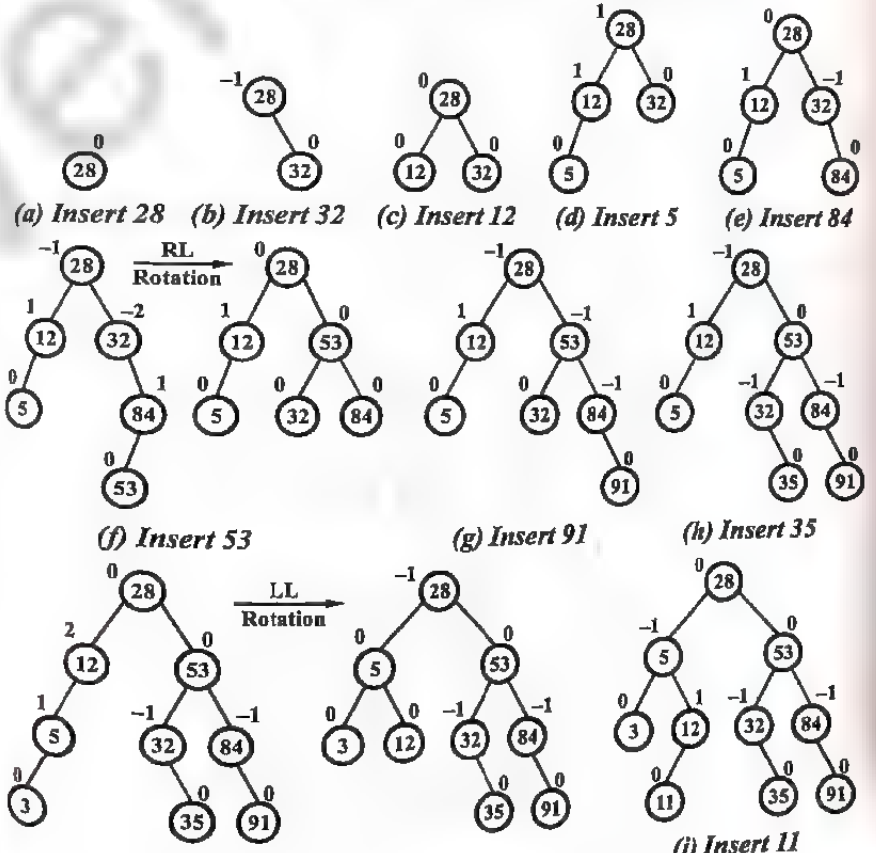


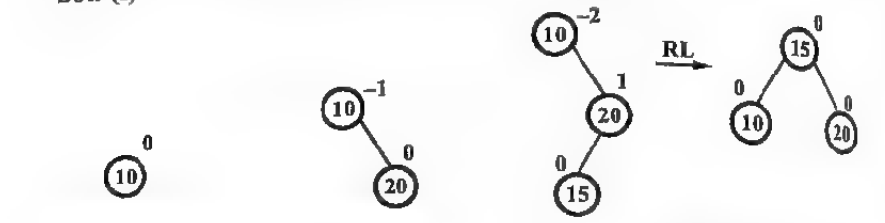
Fig. 5.21

Prob.6. (i) Insert the following sequence of elements into an AVL tree, starting with an empty tree –

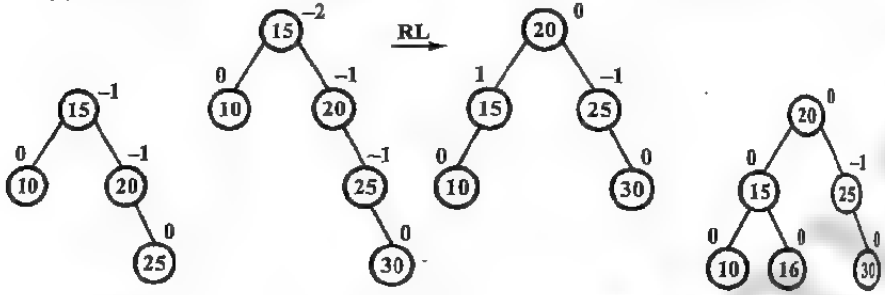
10, 20, 15, 25, 30, 16, 18, 19

(ii) Delete 30 in the AVL tree that you got and re-balance the tree (if required). ***(R.G.P.V., Nov. 2018)***

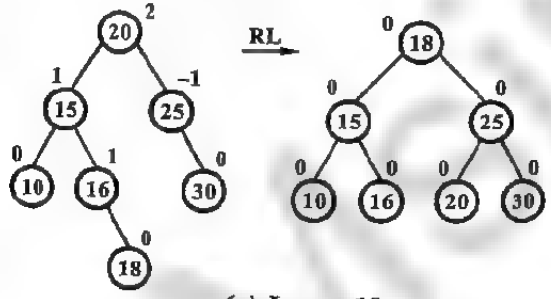
Sol. (i) The construction of AVL tree is as follows –



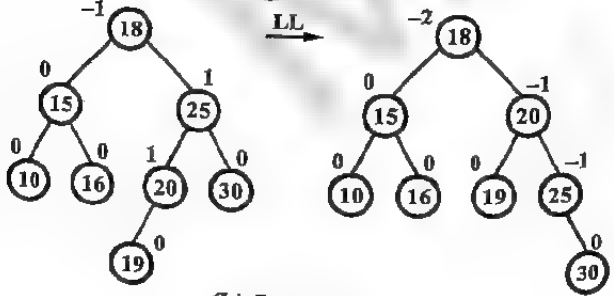
(a) Insert 10 *(b) Insert 20* *(c) Insert 15*



(d) Insert 25 *(e) Insert 30* *(f) Insert 16*

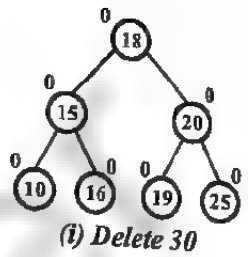


(g) Insert 18



(h) Insert 19

(ii) Delete 30 –



(i) Delete 30

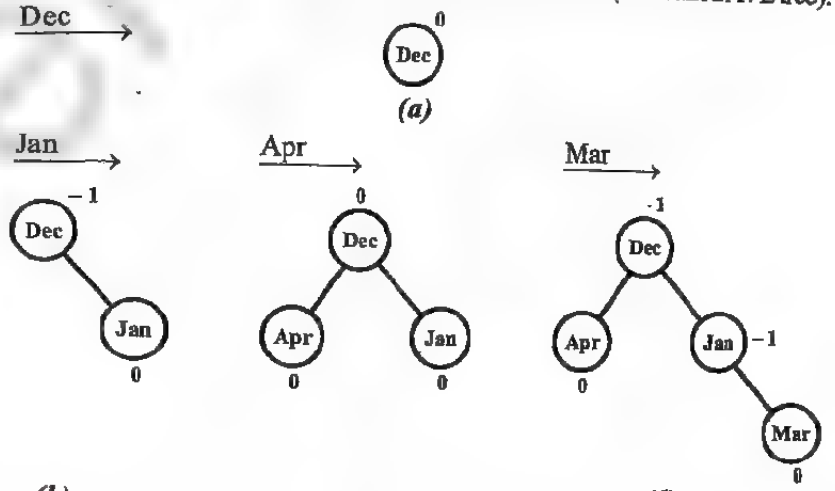
Fig. 5.22

Prob.7. Obtain height balanced trees starting with empty tree on the following set of instructions –

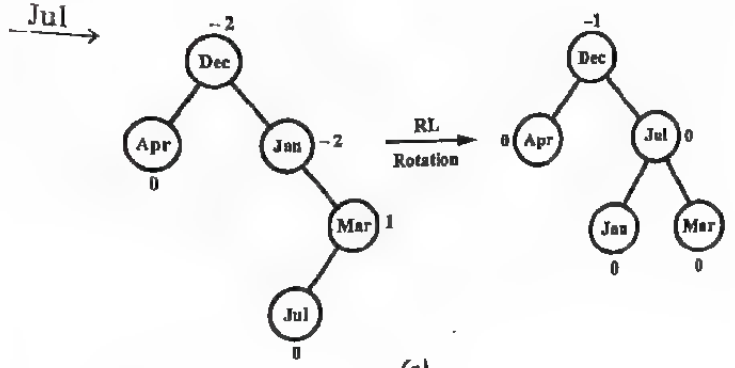
Dec, Jan, Apr, Mar, Jul, Aug, Oct, Feb, Nov, May, June.

(R.G.P.V., Dec. 2008, June 2011, Dec. 2014)

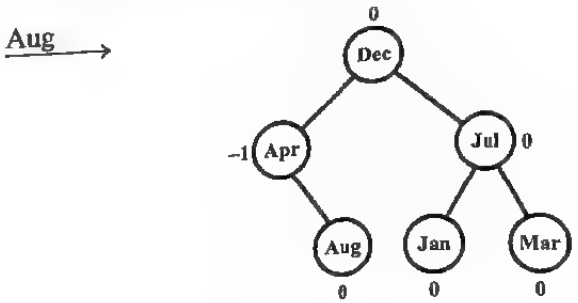
Sol. Fig. 5.23 shows the creation of height balanced tree (also called AVL tree).



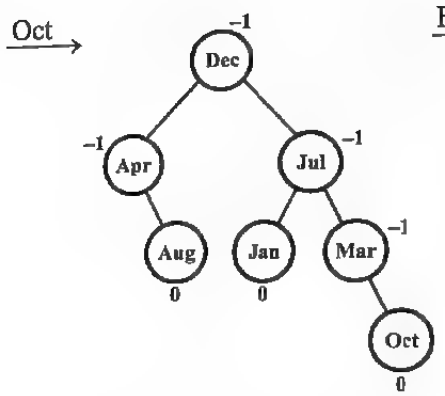
(a) *(b)* *(c)* *(d)*



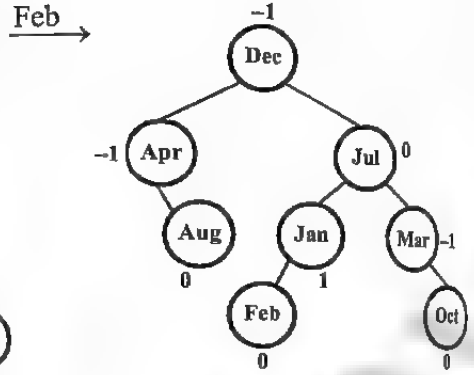
(e)



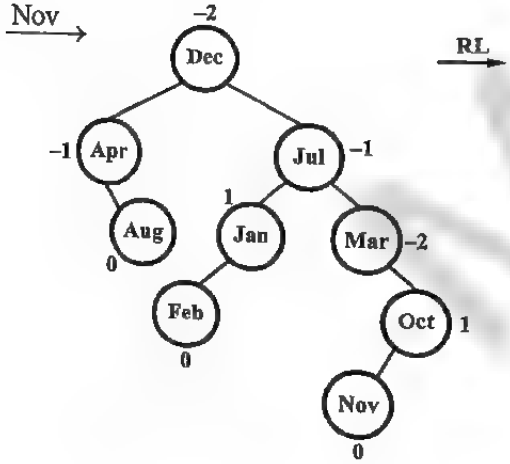
(f)

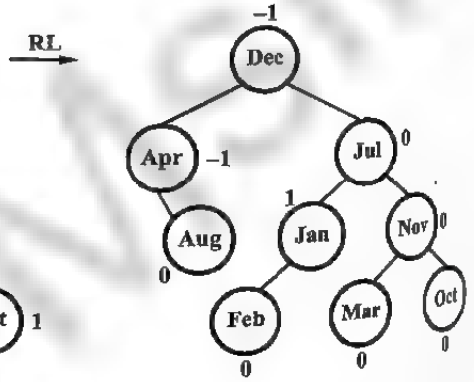


(g)

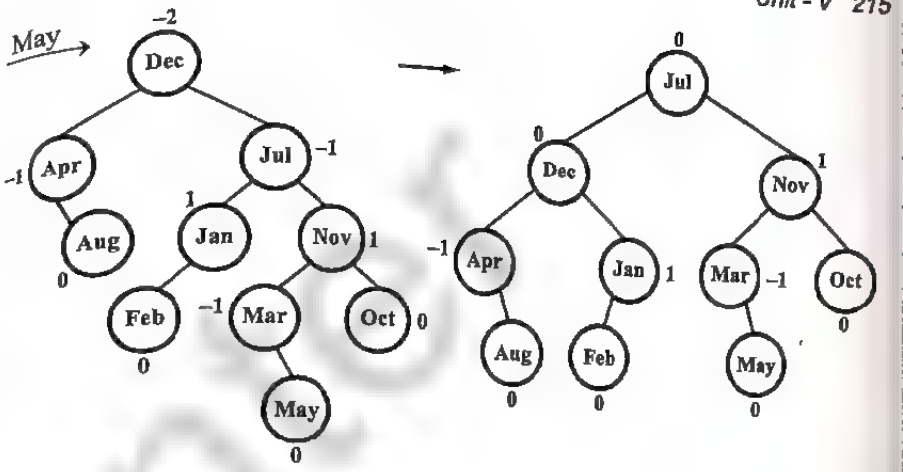


(h)

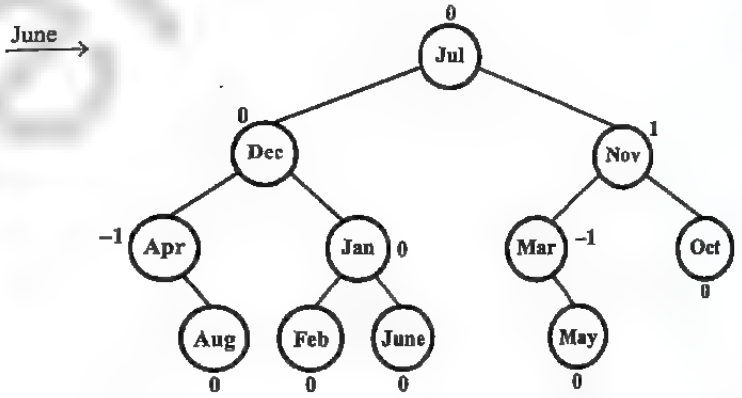




(i)



(j)



(k)

Fig. 5.23

Prob.8. Define height balance tree. Construct a height balance tree starting with empty tree on the following data –

Dec, Jan, Apr, Mar, Jul, Aug, Oct, Feb, Nov, May, Jun.

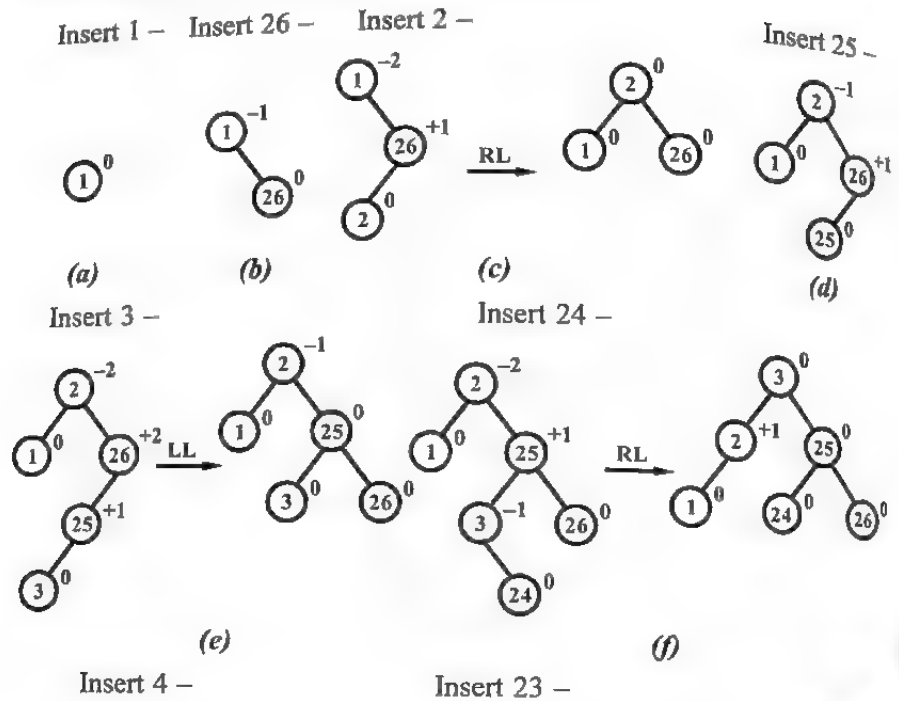
(R.G.P.V., Dec. 2013)

Sol. **Height Balance Tree** – Refer to Q.6.

Problem – Refer to Prob.7.

Prob.9. Insert the elements in the order shown below to build into an AVL tree. Also determine the complexity of this procedure 1, 26, 2, 25, 3, 24, 4, 23, 5, 22, 6. ***(R.G.P.V., June 2014, Dec. 2015)***

Sol. The construction of AVL tree is as follows –



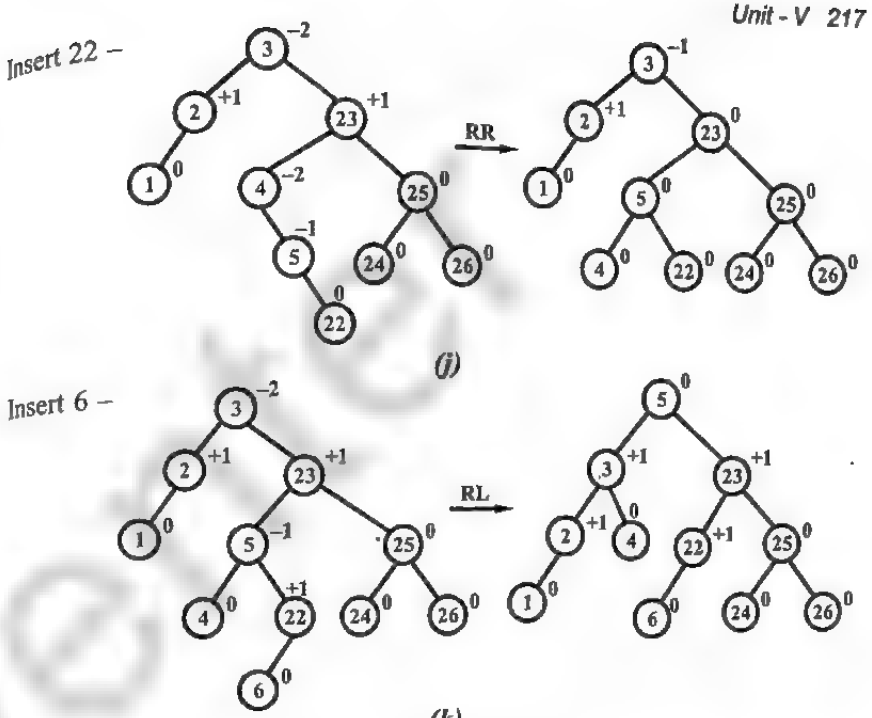


Fig. 5.24

Prob.10. Create a B-tree of order 5 from the following list of data items – 30, 20, 35, 95, 15, 60, 55, 25, 5, 65, 70, 10, 40, 50, 80, 45.
(R.G.P.V., June 2010)

Sol. The order of given B-tree is 5.

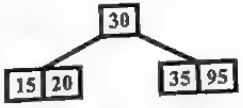
(i) Insert 30 –

(ii) Insert 20 –

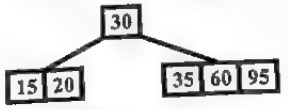
(iii) Insert 35 –

(iv) Insert 95 –

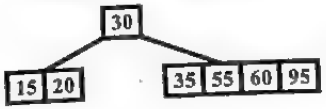
(v) Insert 15 –



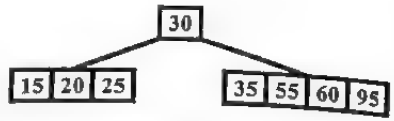
(vi) Insert 60 –



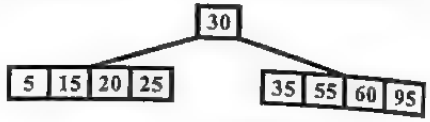
(vii) Insert 55 –



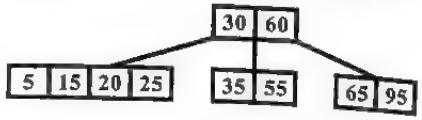
(viii) Insert 25 –



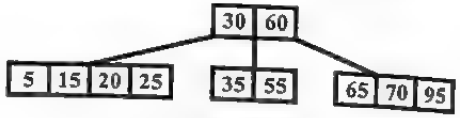
(ix) Insert 5 –



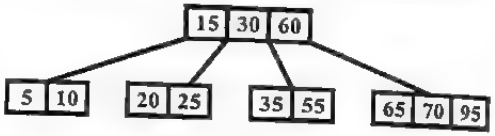
(x) Insert 65 –



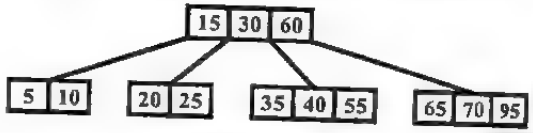
(xi) Insert 70 –



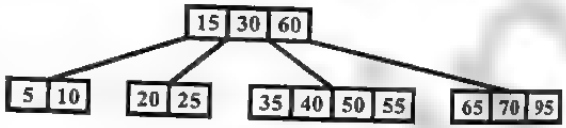
(xii) Insert 10 –



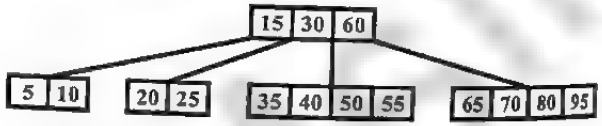
(xiii) Insert 40 –



(xiv) Insert 50 –



(xv) Insert 80 –



(xvi) Insert 45 –

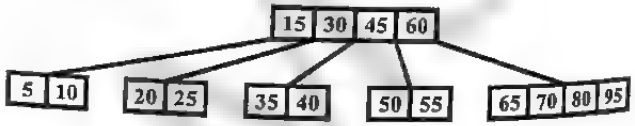


Fig. 5.25

Prob.11. Define B-tree. Insert the entries below, in the order stated, into an initially empty B-tree of order 4.

1, 5, 6, 2, 8, 11, 13, 18, 20, 7, 9 ***(R.G.P.V., Dec. 2012)***

Sol. **B-tree** – Refer to Q.13.

Insertion Operation – The first value 1 is placed in a new node which can accommodate the next two values also i.e.,

Fig. 5.26 (a)

When the fourth value 2 is to be added, the node is split at a median value 5 into leaf nodes with a parent at 5.

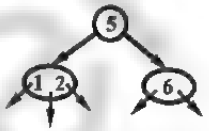


Fig. 5.26 (b)

The following item 8 is to be added in a leaf node. A search for its appropriate place puts it in the node containing 6. Next, 11 is also put in the same. So we have

Fig. 5.26 (c)

Now 13 is to be added. But the right leaf node is full. Therefore, it is split at median value 8 and thus it moves up to the parent. Also it splits up to make two nodes. So we have

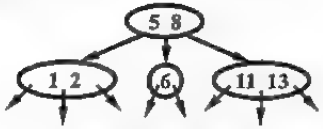


Fig. 5.26 (d)

The remaining items may also be added following the above procedure. The final result is

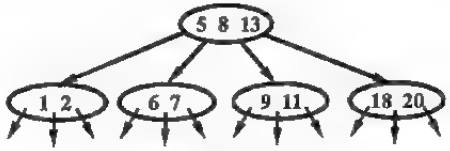


Fig. 5.26 (e)

Prob.12. Create a B-tree for the following list of elements –

L = {86, 50, 40, 3, 94, 10, 70, 90, 110, 113, 116}

given minimization factor t = 3, minimum degree = 2 and maximum degree = 5.

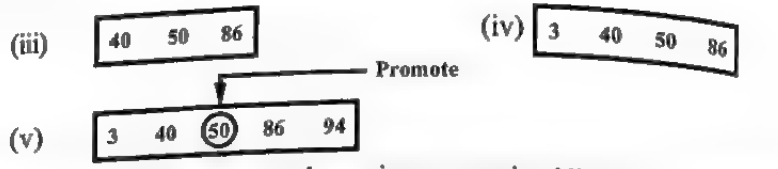
(R.G.P.V., Dec. 2008, June 2013, 2014)

Sol. The order of B-tree is 5. So maximum permitted data items are 5 – 1 = 4 and minimum data item at nonroot node should be (5 – 1)/2 = 2.

Now we start constructing B-tree by inserting elements one by one.

(i) 

(ii) 



Here, the number of items exceeds maximum permitted limited, so break the node.

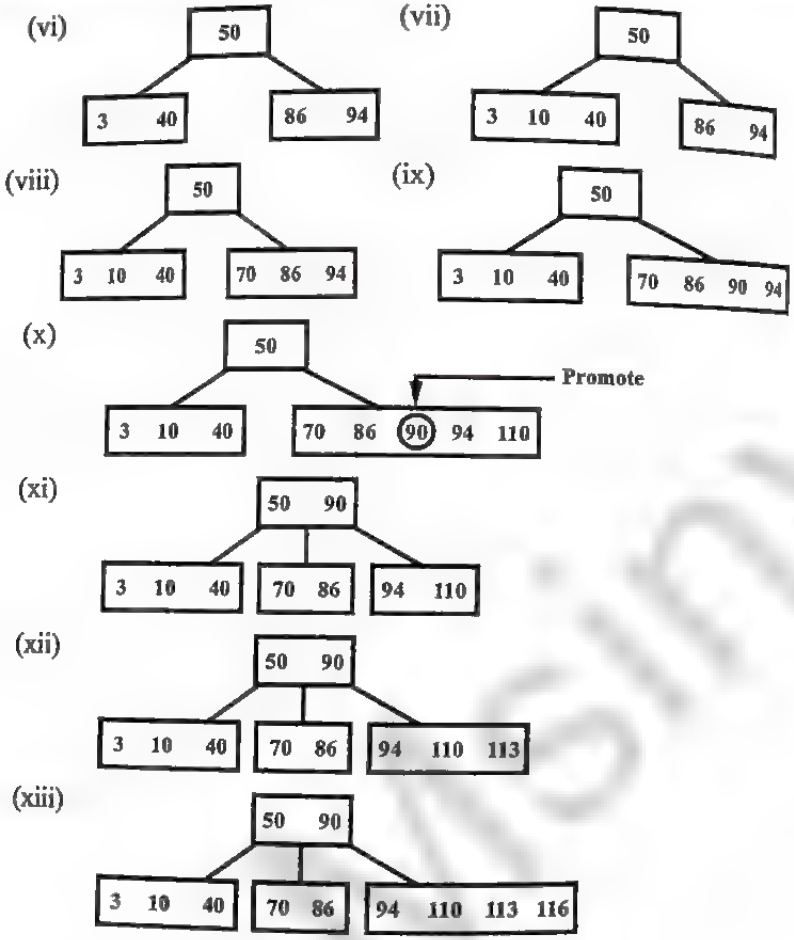


Fig. 5.27

Prob.13. Create a B-tree of order 5 for the following data items – D, H, K, J, B, P, Q, E, A, S, W, T, C, L, N, Y, M.

(R.G.P.V., Dec. 2013)

Sol. The order of given B-tree is 5.

(i) Insert D –

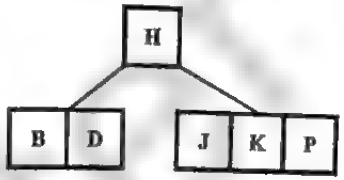
(ii) Insert H –

(iii) Insert K –

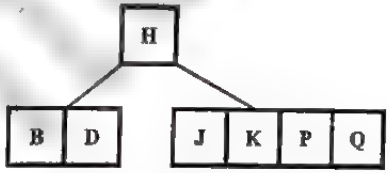
(iv) Insert J

(v) Insert B –

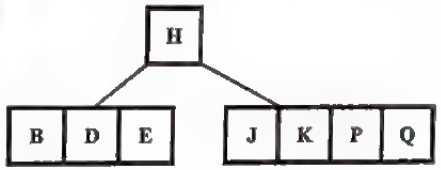
(vi) Insert P –



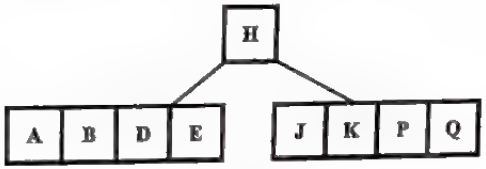
(vii) Insert Q –



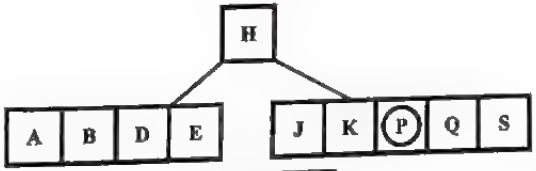
(viii) Insert E –



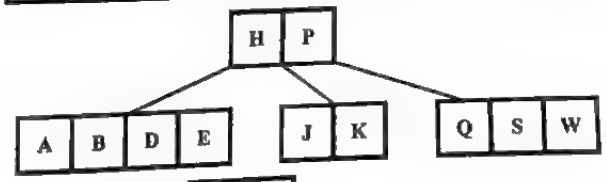
(ix) Insert A –



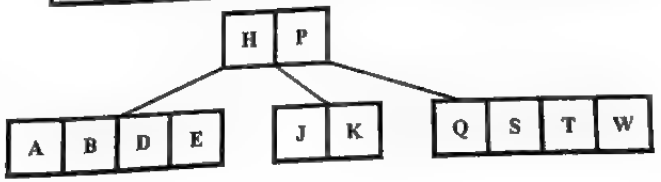
(x) Insert S –



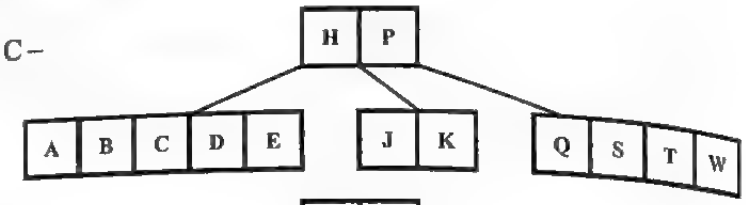
(xi) Insert W –



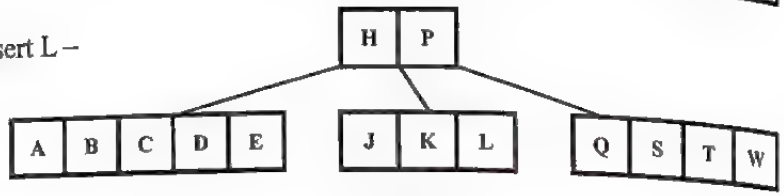
(xii) Insert T –



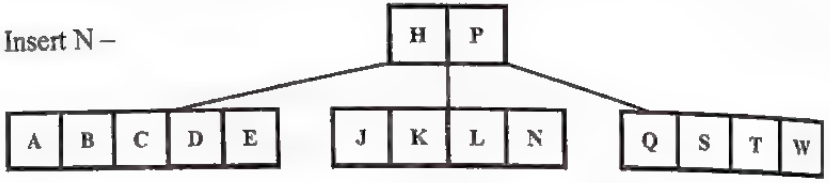
(xiii) Insert C –



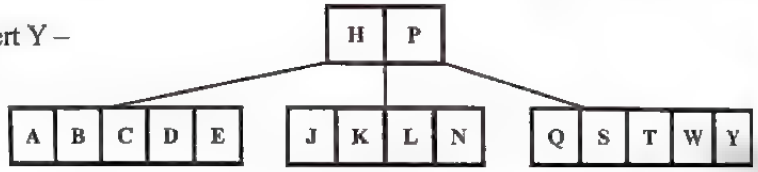
(xiv) Insert L –



(xv) Insert N –



(xvi) Insert Y –



(xvii) Insert M –

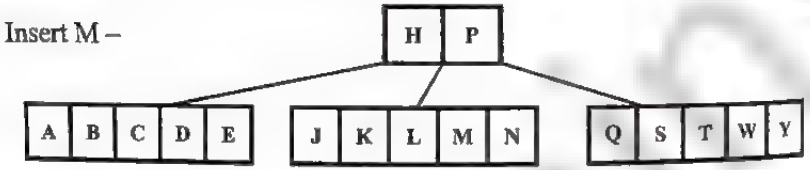


Fig. 5.28

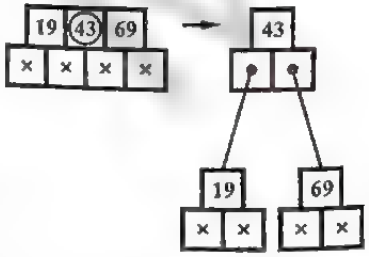
Prob.14. Construct a B-tree of order 3 for the following set of input data– 69, 19, 43, 16, 25, 40, 132, 100, 145, 7, 15, 18.

(R.G.P.V., Dec. 2016)

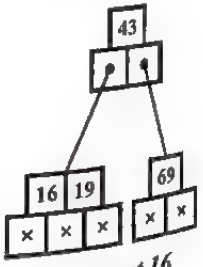
Sol.

(a) Insert 69

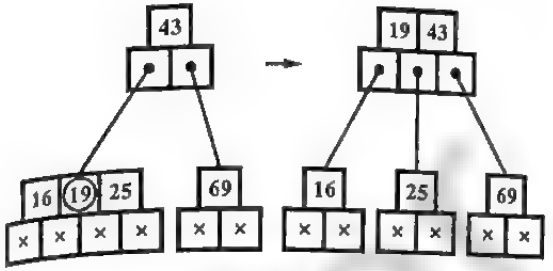
(b) Insert 19



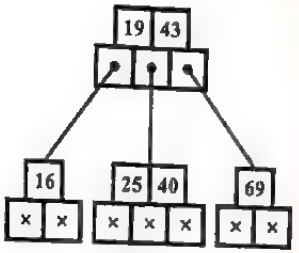
(c) Insert 43



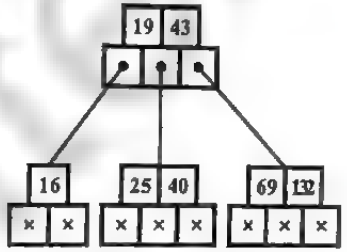
(d) Insert 16



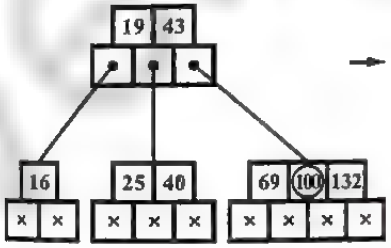
(e) Insert 25

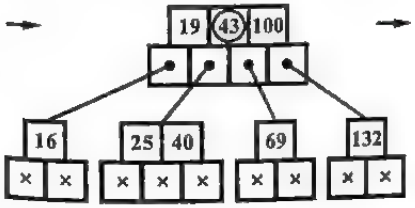


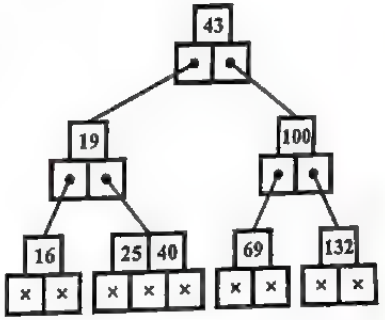
(f) Insert 40



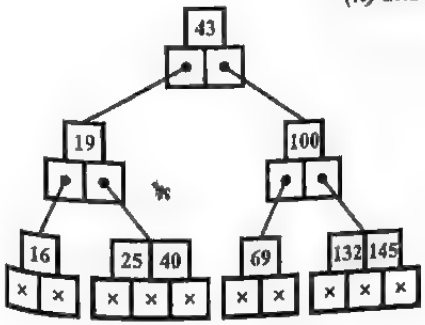
(g) Insert 132



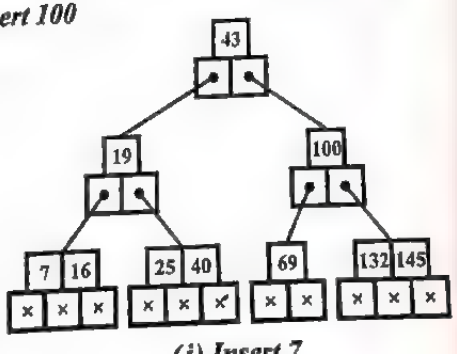




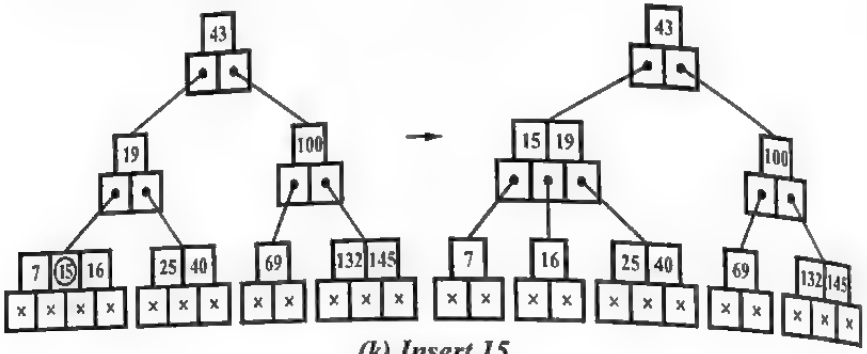
(h) Insert 100



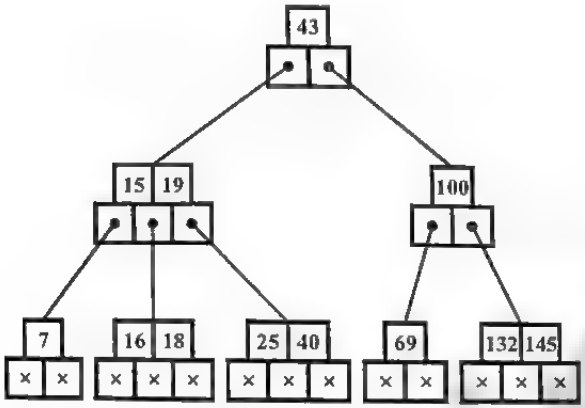
(i) Insert 145



(j) Insert 7



(k) Insert 15



(l) Insert 18

Fig. 5.29

BASIC SEARCH AND TRAVERSAL TECHNIQUES FOR TREES AND GRAPHS (INORDER, PREORDER, POSTORDER, DFS, BFS), NP-COMPLETENESS

Q.21. Write short note on tree traversals. ***(R.G.P.V., May 2019)***

Ans. Tree traversal is one of the most common operations on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner.

While traversing trees, once we start from the root, there are two ways to go, either left or right. At a given node, there are three things to do in some order. To visit the node itself, to traverse its left subtree and to traverse its right subtree. Depending on whether we traverse the node itself before traversing either subtree, between the subtrees or after traversing both subtrees,

there are many traversal orders. If root, left subtree and right subtree are designated by R', L, R respectively, then the possible traversal orders can be

R' L R, R' R L, L R R', LR'R, RLR', RR'L

But conventionally, it is desirable to traverse the left subtree before the right subtree. Therefore, the three standard traversal orders are –

(i) Preorder Traversal (R'LR) – If the root node is visited before traversing its subtrees, it is called the preorder traversal.

(ii) Postorder Traversal (LRR') – If the root node is visited after traversing its subtrees, it is called the postorder traversal.

(iii) Inorder Traversal (LR' R) – If the root node is visited in between the subtrees, it is called the inorder traversal.

Q.22. Write and explain inorder traversal algorithm.

Ans. Algorithm for inorder traversal is given as follows –

Algorithm 5.6 Inorder Traversal

```
treenode = record
{
    Type data; // Type is the data type of data.
    treenode *lchild; treenode *rchild;
}
1.  Algorithm InOrder(t)
2.  // t is a binary tree. Each node of t has
3.  // their fields: lchild, data and rchild.
4.  {
5.      if t ≠ 0 then
6.      {
7.          InOrder (t → lchild);
8.          Visit(t);
9.          InOrder(t → rchild);
10.     }
11. }
```

Here, left child of a node is denoted by lchild, right child is denoted by rchild and the node to be visited is denoted by treenode.

In this algorithm left node of a child is traversed first then the root node is visited and finally the right child of node is traversed. Steps to be followed in this type of traversal are –

(i) Traverse the left subtree of R in inorder.
(ii) Process the root R.
(iii) Traverse the right subtree of R in inorder.

For example, inorder traversal in fig. 5.30 can be shown as follows –

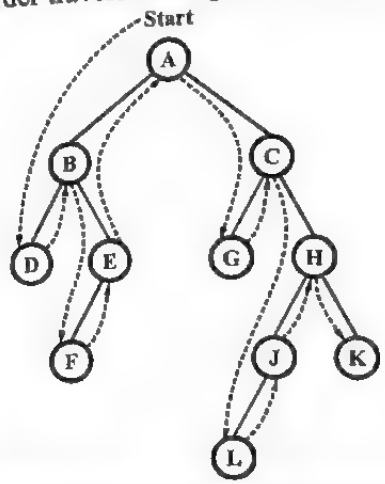


Fig. 5.30 Inorder Traversal of a Tree

Inorder traversal is,

D B F E A G C L J H K

Explanation – In this traversal first traversing starts from root node i.e., node A. We visit first leftchild of root node i.e., node B.

Further we process leftchild of node B, before traversing node B i.e., node D. As node D does not have any children, so it is executed first. After executing left child node itself is executed so we process node B. After visiting node, rightchild of node is visited i.e., node E.

But before executing node E, its left child has to be executed i.e., node F.

After executing leftchild node E is executed. Now left subtree of node A is fully executed, we visit node A.

After executing node A right subtree of node A is visited in same manner.

Q.23. Write and explain preorder traversal algorithm.

Ans. In preorder traversal (DLR), first the root is visited, then left child of node is visited and then rightchild of node is visited.

Steps to be followed in this traversal are as follows –

(i) Process the root R.

(ii) Traverse the left subtree of R in preorder.

(iii) Traverse the right subtree of R in preorder.

Algorithm is given in algorithm 5.7.

Algorithm 5.7 Preorder Traversal

```
1.  Algorithm PreOrder(t)
2.  // t is a binary tree. Each node of t has
3.  // three fields: lchild, data and rchild.
```



```
4.  {
5.      if t ≠ 0 then
6.      {
7.          Visit(t);
8.          PreOrder(t → lchild);
9.          PreOrder(t → rchild);
10.     }
11. }
```

Preorder traversal for fig. 5.31 can be shown as –

A, B, D, E, F, C, G, H, J, L, K.

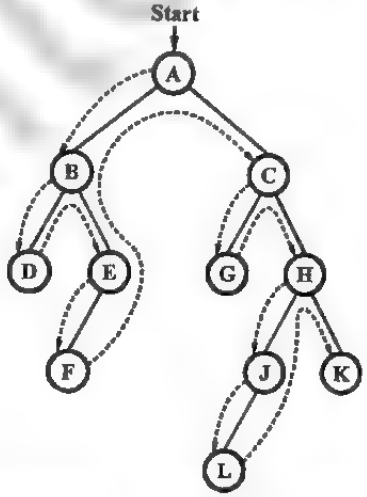


Fig. 5.31 Preorder Traversal a Tree

Explanation – In this traversal, traversing starts from root node of tree i.e., A. First we process the node then its leftchild so, first of all node A is processed. After that left child of node A is to be visited so, here we traverse leftchild of A i.e., B. After processing B, leftchild of B i.e., node D is processed as we do not have any children of D so, we proceed to rightchild of node B i.e., node E. Process node E and visit leftchild of node E i.e., node F. Here, again we do not have any children of node F. So, we proceed towards rightchild of node E. But rightchild of E is empty subtree so we process right subtree of node A. Right subtree of node A has root C.

So, node C is processed, and we move to leftchild of node C i.e., node G. Node G does not have any children, so after processing node G, we go for rightchild of node C i.e., node H. We proceed node H and proceed towards leftchild of node H i.e., node J.

Here, node J is processed and then leftchild of J i.e., node L is processed. And finally we visit rightchild of node H i.e., node K.

Q.24. Write and explain postorder traversal algorithm.

Ans. In postorder traversal (LRD), first of all the leftchild of node is visited, then the rightchild of node is visited and finally the node itself is visited.

Steps to be followed in this type of traversal are as follows –

(i) Traverse the leftsubtree of R in postorder.

(ii) Traverse the right subtree of R in postorder.

(iii) Process the root R.

Algorithm for this type of traversal can be given as follows –

Algorithm 5.8 Postorder Traversal

```
1.  Algorithm PostOrder(t)
2.  // t is a binary tree. Each node of t has
3.  // three fields: lchild, data and rchild.
4.  {
5.      if t ≠ 0 then
6.      {
7.          PostOrder(t → lchild);
8.          PostOrder(t → rchild);
9.          Visit(t);
10.     }
11. }
```

Postorder traversal for fig. 5.32 can be shown as follows –

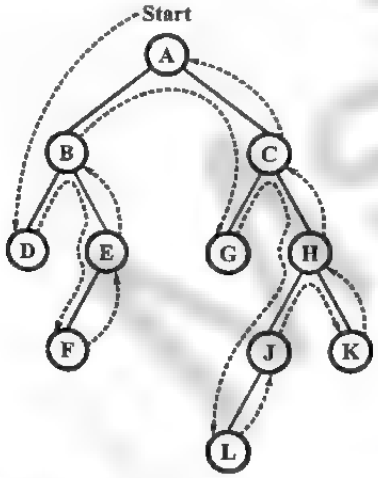


Fig. 5.32 Postorder Traversal a Tree

Post order –

D, F, E, B, G, L, J, K, H, C, A

Explanation – For the postorder traversal first we process leftchild of a node B i.e., node D of node B.

Further, left child of node E i.e., node F is processed and then nodes E and B are processed. Now right subtree of root node A is visited. In this node G is processed first, then nodes L, J, K, H, C and A are processed.

Each traversal can be regarded as a walk through the binary tree. During this walk, each node is reached three times – once from its parent, once on returning its left subtree, and once on returning from its right subtree. In each of these three times a constant amount of work is done, so the total time taken by the traversal is $\theta(n)$. The only additional space needed is that for the recursion stack. If t has depth d, then this space is $\theta(d)$. For an n-node binary tree, $d \leq n$ and so $S(n) = O(n)$.

Q.25. Define preorder, inorder and postorder traversal. Also explain in detail all the traversals. ***(R.G.P.V., June 2015)***

Ans. Refer to Q.23, Q.22 and Q.24.

Q.26. Illustrate the inorder, preorder and postorder traversals for the following binary search tree.

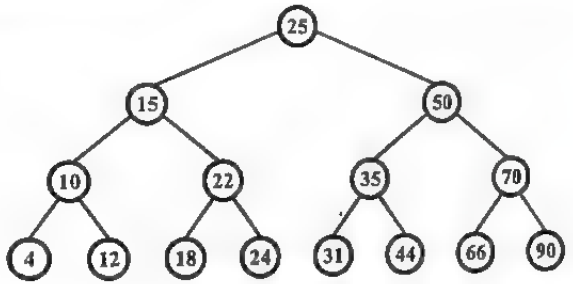


Fig. 5.33

(R.G.P.V., Nov. 2018)

Ans. (i) Inorder – 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90.

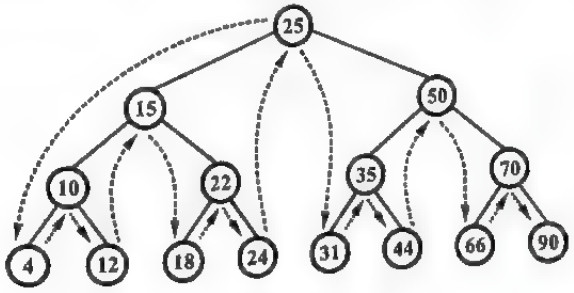


Fig. 5.34

(ii) Preorder – 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90.

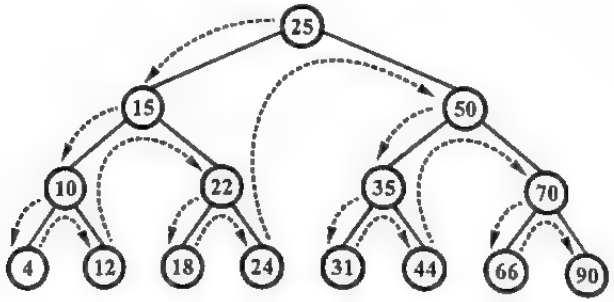


Fig. 5.35

(iii) Postorder – 4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25.

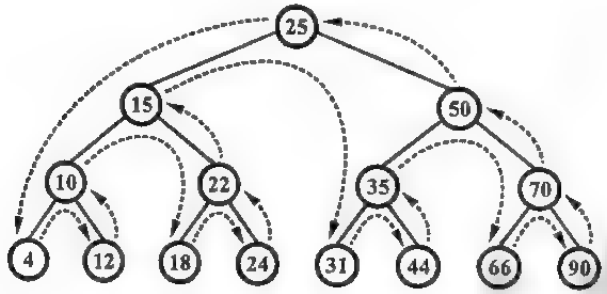


Fig. 5.36

Q.27. Show preorder, inorder and postorder for the following tree –

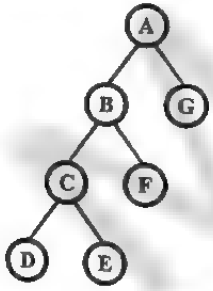


Fig. 5.37 ***(R.G.P.V., June 2017)***

Ans. (i) Preorder – A B C D E F G

(ii) Inorder – D C E B F A G

(iii) Postorder – D E C F B G A

Q.28. Describe the graph traversal techniques.

Or

Write short note on graph traversal. ***(R.G.P.V., Dec. 2017)***

Ans. In many problems, we like to investigate all the vertices in a graph in some systematic order, just as with binary trees, where we developed several systematic traversal methods. Generally in tree traversal, we had a root vertex with which we started; in graphs, we often do not have any one vertex singled out as special, and therefore the traversal may start at an arbitrary vertex. Although there are many possible orders for visiting the vertices of the graph, two methods are of particular importance. There are two traversal techniques in graphs –

(i) Depth-first Traversal – Depth-first traversal of a graph is roughly similar to preorder traversal of an ordered tree. Suppose that the traversal has just arrived a vertex V, and suppose w_1, w_2, ..., w_k be the vertices adjacent to V. Then we shall next visit w_1 and keep w_2, ..., w_k waiting. After visiting w_1, we traverse all the vertices to which it is adjacent before returning to traverse w_2, ..., w_k.

(ii) Breadth-first Traversal – This traversal of a graph is roughly similar to level-by-level traversal of an ordered tree. If the traversal has just visited a vertex V, then it next visits all the vertices adjacent to V, putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to V have been visited. Fig. 5.38 explains the order of visiting the vertices of one graph under both depth-first and breadth-first traversals.

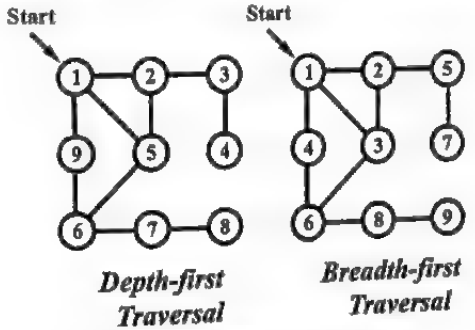


Fig. 5.38 Graph Traversal

Q.29. Compare DFS and BFS. ***(R.G.P.V., June 2016, May 2018)***

Ans. The comparison between DFS and BFS is given below –

(i) In depth first search start vertex V is visited first and if w_1, w_2, ..., w_k be the vertices adjacent to V, then the vertex w_1 is visited next. After visiting w_1 all the vertices adjacent to w_1 are visited in depth first manner

before returning to traverse w_1, ..., w_k. While breadth first search is similar to level-by-level traversing of an ordered tree.

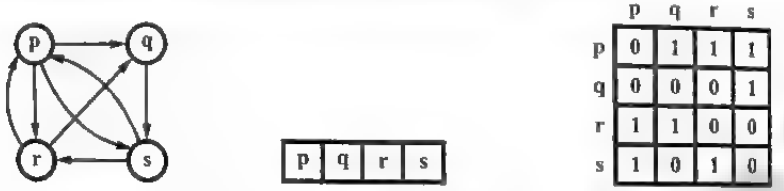
(ii) Depth first search is similar to preorder traversal of an ordered tree, while breadth first search is similar to inorder traversal.

(iii) Depth first search uses a stack to hold nodes that are waiting to be processed, while breadth first search uses a queue.

Q.30. Write any two data structures that are suitable for representing a graph. Write an algorithm for depth first traversal of a graph using one of your two data structures. ***(R.G.P.V., Dec. 2016)***

Ans. Graph can be represented by using an array or linked list.

(i) Array Representation of Graph – The nature of array representation of a graph is sequential, because vertices are stored sequentially in it. Graph implementation requires a one-dimensional array to store data and a two-dimensional array to store its connectivity with other data. Generally, two-dimensional array represents the adjacency matrix.



(a) Graph ***(b) 1-D Representation*** ***(c) 2-D Representation***

Fig. 5.39 Array Representation of Graph

The syntax to implement graph using array is –

```
Struct graph
{
    char V[max];          // vertex implementation
    int e[max] [max];     // edge implementation
}
```

(ii) Linked List Representation of a Graph – The linked list representation of a graph is also termed as adjacency structure.

Linked representation of graph in fig. 5.39 (a) is shown in fig. 5.40.

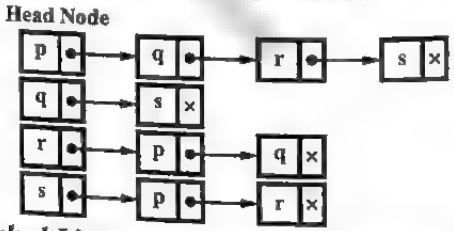


Fig. 5.40 Linked List Representation of Graph in Fig. 5.39 (a)

Algorithm of Depth First Traversal (When Graph is Represented using Linked List) –

Algorithm LDFT (head, visited[n])

/* This algorithm uses linked list having two elements DATA and LINK. Linked list is the representation of graph and head is the node of graph from which traversal starts and visited is an integer or boolean array of size n. Where, n is the number of vertices in graph. Initially all the values of visited[ ] is 0 */

```
Initialize pointer P = head;
print(head → data);
visited [head] = 1;
while (P → link is not NULL) do
P = P → link;
if (visited [P] = 0) then
        call LDFT (P, visited);
    end while
end LDFT
```

Q.31. Write down the algorithm of BFS and DFS of a tree. How do you maintain the open and closed list in the algorithm if the nodes which are explored and whose children has not been generated will be in the open list and the nodes which are explored and whose children has been generated in the closed list ? Taking suitable example of tree, explain these list manipulations in BFS and DFS algorithms.

Or

Write an algorithm for BFS and DFS. ***(R.G.P.V., Dec. 2009)***

Or

What is BFS and DFS ? Explain with suitable example with respect to tree. ***(R.G.P.V., Dec. 2015)***

Ans. **Breadth-first Search** – Breadth-first Search explores the space level by level only when there are no more states to be explored at a given level does the algorithm move on to the next level.

We implement BFS using lists open & closed to keep track of progress through the state space. In the open list, the elements will be, who have been generated but whose children has not been examined. Closed list records the states that have been examined and whose children has been generated. The order of removing the states from open list will be the order of searching. Open is maintained as queue on first in first out data structure. States are added to the right of the list and removed from the left.

The algorithm for Breadth-first search procedure is as follows –

Algorithm 5.9 Breadth-first-Search

```
begin
    open = [start] ;
    closed = [ ] ;
    while open ≠ [ ] do
    begin
    remove leftmost state from open call it x ;
    if x is a goal then return success
    else
    begin
        generate children of x ;
        put x on closed ;
        put children on right end of open ;
    end
    end
    return (failure)
end
```

Depth-first Search – In depth first search when a state is examined all of its children and their descendants are examined before any of its siblings. Depth first search goes deeper into the search space whenever this is possible, only when no further descendants of a state can be found, are its siblings considered.

The algorithm for Depth-first search is as follows –

Algorithm 5.10 Depth-first Search

```
begin
    open = [start] ;
    closed = [ ] ;
    while open ≠ [ ] do
    begin
        remove left most state from open call it x ;
        if x is a goal then return success
    else
    begin
        generate children of x ;
        put x on closed ;
        put children on left end of open ;
    end
    end
    return (failure)
end.
```


For example, consider the tree shown in fig. 5.41. Here the path followed by BFS is shown by dotted lines. Fig. 5.41 explains BFS procedure.

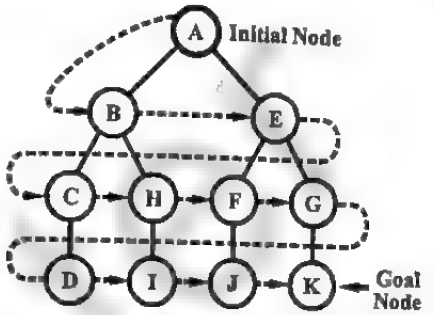


Fig. 5.41 Breadth-first Search

The open and closed lists maintained by BFS are shown below –

open = [A] ;	closed = [ ]
open = [B, E] ;	closed = [A]
open = [E, C, H] ;	closed = [A, B]
open = [C, H, F, G] ;	closed = [A, B, E]
open = [H, F, G, D] ;	closed = [A, B, E, C]
open = [F, G, D, I] ;	closed = [A, B, E, C, H]
open = [G, D, I, J] ;	closed = [A, B, E, C, H, F]
open = [D, I, J K] ;	closed = [A, B, E, C, H, F, G]
open = [I, J, K] ;	closed = [A, B, E, C, H, F, G, D]
open = [J, K] ;	closed = [A, B, E, C, H, F, G, D, I]
open = [k] ;	closed = [A, B, E, C, H, F, G, D, I, J]
open = [ ] ;	closed = [A, B, E, C, H, F, G, D, I, J, K]

To understand DFS, consider the tree below. Here dotted lines show the path followed by DFS. Fig. 5.42 explains DFS procedure.

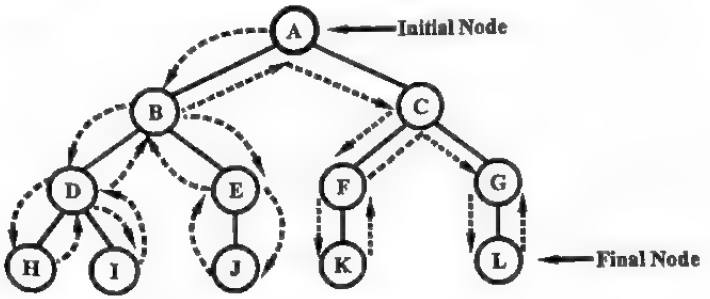


Fig. 5.42 Depth-first Search

The open and closed list maintained by DFS is shown below –

open = [A] ;	closed = [ ]
open = [B, C]	closed = [A]
open = [D, E, C] ;	closed = [A, B]
open = [H, I, E, C] ;	closed = [A, B, D]
open = [I, E, C] ;	closed = [A, B, D, H]
open = [E, C] ;	closed = [A, B, D, H, I]
open = [J, C] ;	closed = [A, B, D, H, I, E]
open = [C] ;	closed = [A, B, D, H, I, E, J]
open = [F, G] ;	closed = [A, B, D, H, I, E, J, C]
open = [K, G] ;	closed = [A, B, D, H, I, E, J, C, F]
open = [G] ;	closed = [A, B, D, H, I, E, J, C, F, K]
open = [L] ;	closed = [A, B, D, H, I, E, J, C, F, K, G]
open = [ ] ;	closed = [A, B, D, H, I, E, J, C, F, K, G, L]

Q.32. List out the techniques for traversals in graph. Also explain each in detail with its procedure. ***(R.G.P.V., June 2015)***

Ans. Refer to Q.28 and Q.31.

Q.33. Write BFS algorithm and analyse the running time of algorithm. ***(R.G.P.V., Dec. 2008, 2010)***

Ans. **BFS Algorithm** – Refer to Q.31.

Analysis of Running Time – After initialization, each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take O(1) time, so the total time devoted to queue operations is O(v). Because the adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\theta(E)$, at most O(E) time is spent in total scanning adjacency lists. The overhead for initialization is O(v), and thus the total running time of BFS is O(V + E). Thus, breadth-first search runs in time linear in the size of the adjacency list representation of G.

Q.34. Write DFS algorithm and calculate the running time of it. ***(R.G.P.V., Dec. 2011)***

Ans. **DFS Algorithm** – Refer to Q.31.

Running Time of DFS – A depth first traversal takes O(N*E) time for adjacency list representation and $O(N^2)$ for matrix representation.

Q.35. Write BFS and DFS algorithms and also analyse the running time of algorithm. ***(R.G.P.V., June 2011, 2013)***

Ans. **BFS and DFS Algorithm** – Refer to Q.31.

Running Time of BFS – Refer to Q.33.

Running Time of DFS – Refer to Q.34.

Q.36. Starting from vertex V_4 apply BFS and DFS in the given ahead fig. 5.43.

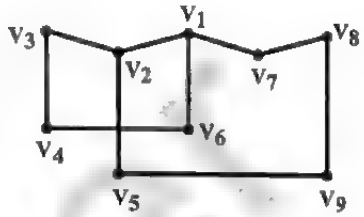


Fig. 5.43

(R.G.P.V., June 2010)

Ans. BFS for the fig. 5.43 is given by –

$V_4\ V_3\ V_6\ V_2\ V_1\ V_5\ V_7\ V_9\ V_8$

DFS for the given fig. 5.43 is –

$V_4\ V_6\ V_1\ V_7\ V_8\ V_9\ V_5\ V_2\ V_3$

Q.37. Explain undecidable problem. ***(R.G.P.V., June 2015)***

Ans. An undecidable problem is a problem for which there is no algorithm that can solve it. Alan Turing proved that the famous halting problem is undecidable. The halting problem can be simply stated as follows – Given an input and a Turing machine, there is no algorithm to determine if the machine will eventually halt. There are several problems in mathematics and computer science that are undecidable.

Q.38. Discuss polynomial and non-polynomial time algorithms. ***(R.G.P.V., June 2016)***

Ans. If the complexity of an algorithm is expressed as O(p(n)) where p(n) is some polynomial of n, then the algorithm is said to be a polynomial time algorithm. Generally, polynomial time algorithms are tractable. Any algorithm with a time complexity that cannot be bounded by such bound, is called a non-polynomial time algorithms.

Q.39. What are deterministic and non-deterministic algorithms ? How these can be used to explain NP-completeness and NP-hardness ? ***(R.G.P.V., June 2007)***

Ans. Algorithms with the property that the result of every operation is uniquely defined, are termed ***deterministic algorithms.*** In a theoretical framework, we can remove this restriction on the outcome of every operation. We can allow algorithms to contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities. The machine executing such operations is allowed to choose any one of these outcomes subject to a

termination condition to be defined later. This leads to the concept of a ***nondeterministic algorithm.*** Three new functions specify such algorithms –

(i) Choice (S) arbitrarily chooses one of the elements of set S.

(ii) Failure( ) signals an unsuccessful completion.

(iii) Success( ) signals a successful completion.

The assignment statement x: = Choice (1, n) could result in x being assigned any one of the integers in the range [1, n]. There is no rule specifying how this choice is to be made. The Failure( ) and Success( ) signals are used to define a computation of the algorithm. These statements cannot be used to effect a **return.** Whenever there is a set of choices that leads to a successful completion, then one such set of choices is always made and the algorithm terminates successfully. A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal. The computing times for Choice, Success and Failure are taken to be O(1). A machine capable of executing a nondeterministic algorithm in this way is called a ***nondeterministic machine.***

Q.40. Define P and NP problems.

Or

Explain class P problem. ***(R.G.P.V., June 2015)***

Ans. P is the set of all decision problems solvable by deterministic algorithms in polynomial time. NP is the set of all decision problems solvable by non-deterministic algorithms in polynomial time.

Q.41. Prove that P ⊆ NP.

Ans. Since deterministic algorithms are just a special case of nondeterministic ones, so we conclude that P ⊆ NP. Fig. 5.44 displays the relationship between P and NP assuming that P ≠ NP.

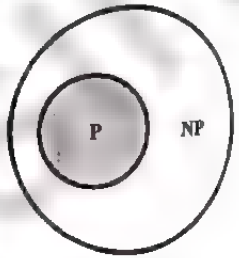


Fig. 5.44

The language verified by a verification algorithm A is

L = {x ∈ {0, 1}*; There exists y ∈ {0, 1}* such that A (x, y) = 1}

if L ∈ P, then L ∈ NP, since if there is a polynomial-time algorithm to decide L, the algorithm can be easily converted to a two argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in L. Thus, P ⊆ NP.

Q.42. What is P, NP class problems ? Explain the concept with suitable example. ***(R.G.P.V., June 2014, Dec. 2014)***

Ans. Refer to Q.40 and Q.41.

Q.43. What is the relationship between NP-hard and NP-complete classes ?

Or

Explain NP-hard problems. ***(R.G.P.V., June 2015)***

Ans. A problem L is NP-hard if and only if satisfiability reduce to L (satisfiability ∝ L). A problem is NP-complete if and only if L is NP-hard and L ∈ NP.

Only a decision problem can be NP-complete. However, an optimization problem may be NP-hard. Furthermore if L_1 is a decision problem and L_2 an optimization problem, then it is possible that $L_1 \propto L_2$. One can trivially show that the Knapsack decision problem reduces to the Knapsack optimization problem. For the clique problem one can easily show that the clique decision problem reduces to the clique optimization problem. In fact, one can also show that these optimization problems reduce to their corresponding decision problems. Fig. 5.45 shows the relationship among these classes.

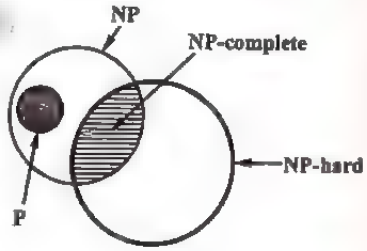


Fig. 5.45

Q.44. Discuss the relationship between class P, NP, NP-complete and NP-hard problems with example of each class.

(R.G.P.V., Dec. 2008, June 2009, Dec. 2011, 2015)

Or

Discuss the relationship between P, NP, NP-complete and NP-hard problems. ***(R.G.P.V., June 2015)***

Ans. Refer to Q.40, Q.41 and Q.43.

Q.45. Write short note on NP-completeness.

(R.G.P.V., Dec. 2006, 2007, June 2017, Nov. 2018, May 2019)

Or

Explain NP-completeness briefly. ***(R.G.P.V., June 2009)***

Or

Explain NP-completeness with example. ***(R.G.P.V., Dec. 2013)***

Or

Write a short note on NP-completeness problem. ***(R.G.P.V., Dec. 2016)***

Or

Discuss in detail NP-complete problem with example.

(R.G.P.V., May 2018)

Ans. Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, within a polynomial-time factor.

This means, if $L_1 \leq_p L_2$, then L_1 is not more than a polynomial factor harder than L_2. Which is why the "less than or equal to" notation for reduction is mnemonic. NP complete problems are the problems whose status are unknown. Some of the examples of the NP-complete problems are –

(i) Travelling Salesman Problem – Given n cities, the distance between them and a number D, does there exist a tour programme for a salesman to visit all the cities so that the total distance travelled is atmost D ?

(ii) Zero-one Programming Problem – Given m simultaneous equations

$$\sum_{j=1}^{m} a_{ij}x_j = b_i \ , \quad i = 1, 2 \ldots, n$$

Does there exist values, zero or one, for x_j so that the above equations are satisfied ?

(iii) Satisfiability Problem – Given a formula involving propositional variables and logical connectives. Does there exist a choice of truth values for the variables for which the given formula assumes the truth value T.

A language $L \subseteq [0, 1]^*$ is **NP-complete** if

1 $L \in NP$, and

2 $L' \leq_p L$ for every $L' \in NP$.

Q.46. Define NP completeness and reducibility of problems. What are NP hard problems ? ***(R.G.P.V., Dec. 2009)***

Or

Explain NP-complete and NP-hard problems. ***(R.G.P.V., June 2010)***

Or

What do you understand by NP-hard and NP-complete classes ? ***(R.G.P.V., Dec. 2012)***

Or

Explain NP-complete and NP-hard problem. ***(R.G.P.V., Dec. 2017)***

Ans. **NP-completeness** – Refer to Q.45.

If a language L satisfies property 2, but not necessarily property 1, we say that language L is **NP-hard.** A problem that is NP-complete has the property that it can be solved in polynomial time if and only if all other NP-complete problems can also be solved in polynomial time. If an NP-hard problems can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time. All NP-complete problems are NP-hard, but some NP-hard problems are not known to be NP-complete.

Q.47. Show that the travelling salesman problem is NP-complete. ***(R.G.P.V., June 2008, 2013)***

Ans. **Proof** – We first show that TSP (Travelling-salesman problem) belongs to NP. Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where $E' = \{(i, j): i, j \in V\}$, and we define the cost function c by

$$c(i, j) = \begin{cases} 0 & \text{if}(i, j) \in E, \\ 1 & \text{if}(i, j) \notin E. \end{cases}$$

The instance of TSP is then (G', c, 0), which is easily formed in polynomial time.

We now show that graph G has a Hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a Hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0. Therefore, h' contains only edges in E. We conclude that h is a Hamiltonian cycle in graph G.

Q.48. Show that the Hamiltonian-path problem is NP-complete. ***(R.G.P.V., June 2008)***

Or

Show that Hamiltonian cycle problem is NP-complete. ***(R.G.P.V., June 2012)***

Ans. We first show that HAM-CYCLE belongs to NP. Given a graph $G = (V, E)$, our certificate is the sequence of $|V|$ vertices that make up the Hamiltonian cycle. The verification algorithm checks that this sequence contains each vertex in V exactly once and that with the first vertex repeated at that end, if forms a cycle in G. This verification can be performed in polynomial time.

We now prove that HAM-CYCLE is NP-complete by showing that 3 CNF-SAT $\leq$ pHAM-CYCLE. Given a 3-CNF boolean formula ϕ over variables

$x_1, x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$ each containing exactly 3 distinct literals, we construct a graph G = (V, E) in polynomial time such that G has a Hamiltonian cycle if and only if ϕ is satisfiable. Our construction is based on ***widgets***, which are pieces of graphs that enforce certain properties.

Our first widget is the subgraph A shown in fig. 5.46 (a). Suppose that A is a subgraph of some graph G and that the only connections between A and the remainder of G are through the vertices a, a', b and b'. Furthermore, suppose that graph G has a Hamiltonian cycle. Since any Hamiltonian cycle of G must pass through vertices z_1, z_2, z_3 and z_4 in one of the ways shown in fig. 5.46 (b) and (c), we may treat subgraph A as if it were simply a pair of edges (a, a') and (b, b') with the restriction that any Hamiltonian cycle of G must include exactly one of these edges. We shall represent widget A as shown in fig. 5.46 (d).

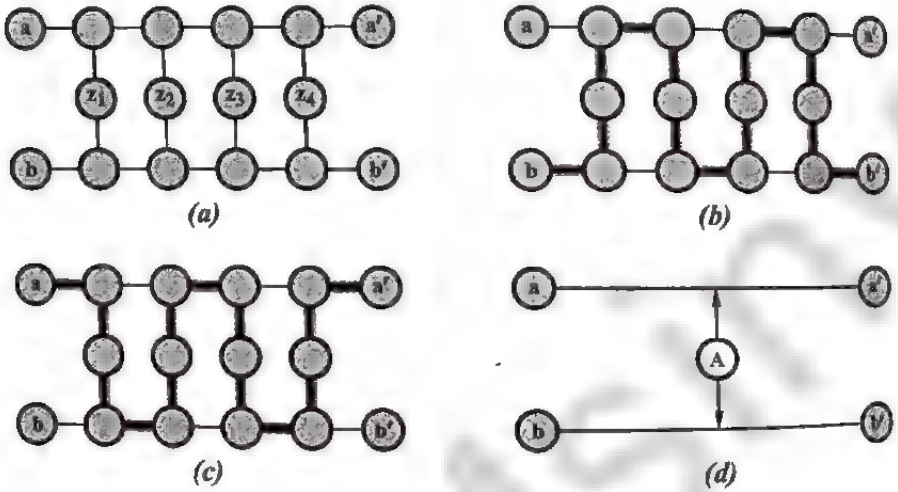


Fig. 5.46

The subgraph B in fig. 5.47 is our second widget. Suppose that B is a subgraph of some graph G and that the only connections from B to the remainder of G are through vertices b_1, b_2, b_3 and b_4. A Hamiltonian cycle of graph G cannot traverse all of the edges (b_1, b_2), (b_2, b_3) and (b_3, b_4), since then all vertices in the widget other than b_1, b_2, b_3 and b_4 would be missed. However, a Hamiltonian cycle of G may traverse any proper subset of these edges. Fig. 5.47 (a) to (e) show five su h subsets; the remaining two subsets can be obtained by performing a top-to-bottom flip of parts (b) and (e). This widget is represented as in fig. 5.47 (f), the idea being that at least one of the paths pointed to by the arrows must be taken by a Hamiltonian cycle.

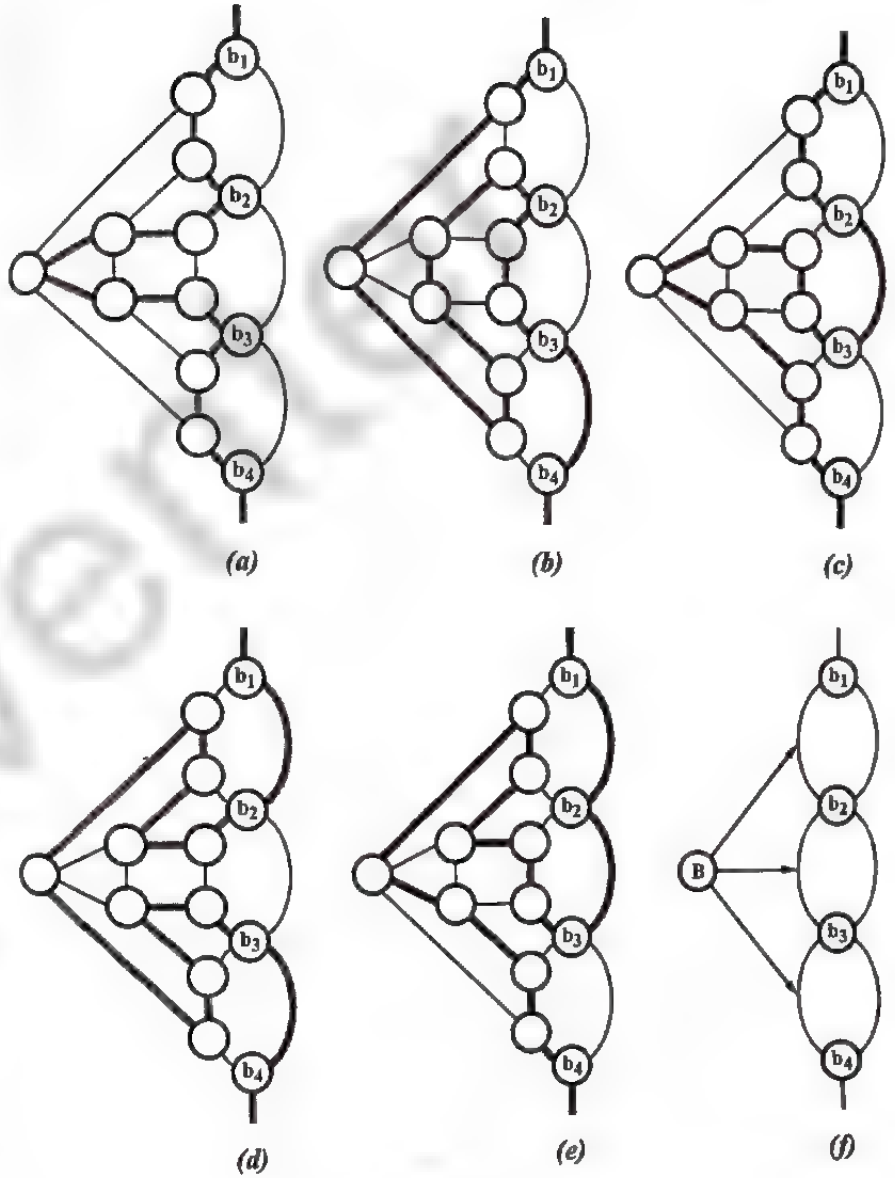


Fig. 5.47

The graph G that we shall construct consists mostly of copies of these two widgets. The construction is shown in fig. 5.48. For each of the k clauses in ϕ, we include a copy of widget B, and we join these widgets together in series as follows. Letting b_{ij} be the copy of vertex b_j in the i^{th} copy of widget B, we connect $b_{i,4}$ to $b_{i+1,1}$ for i = 1, 2, ..., k – 1.

Then for each variable x_m in ϕ, we include two vertices x'_m and x''_m. These two vertices are connected by means of two copies of the edge (x'_m, x''_m), which are denoted by e_m and $\bar{e}_m$. The idea is that if the Hamiltonian cycle takes edge e_m, it corresponds to assigning variable x_m the value 1. If the Hamiltonian cycle takes edge $\bar{e}_m$, the variable is assigned the value 0. Each pair of these edges forms a two-edge loop; these small loops are connected in series by adding edges (x'_m, x''_{m+1}) for $m = 1, 2, ..., n - 1$. The left (clause) side of the graph is connected to the right (variable) side by means of two edges $(b_{1,1}, x'_1)$ and $(b_{k,4}, x''_n)$ which are the topmost and bottommost edges in fig. 5.48.

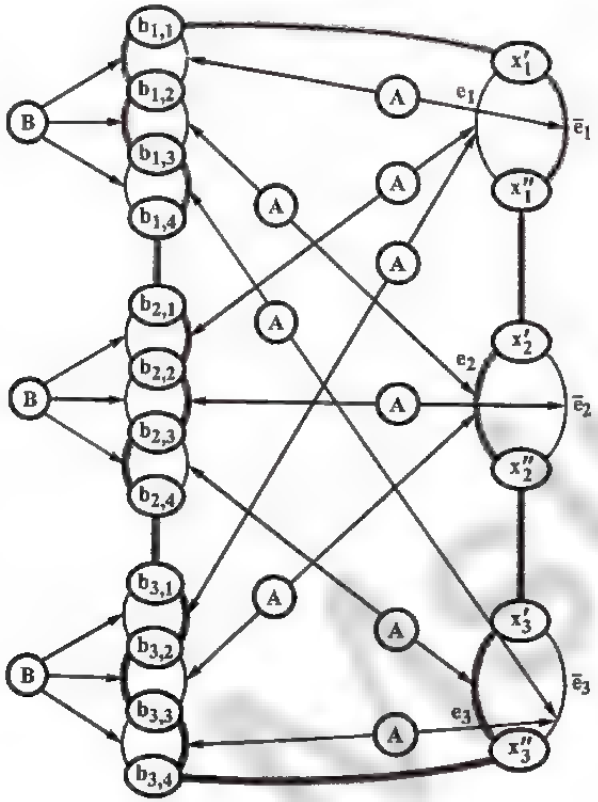


Fig. 5.48

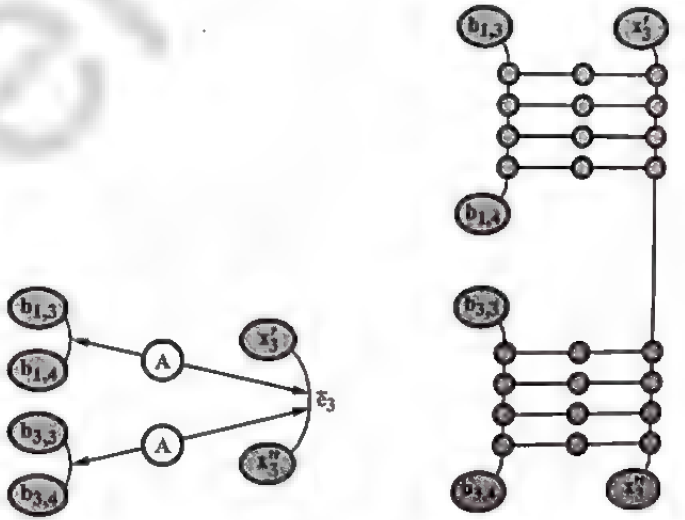
We are not yet finished with the construction of graph G, since we have yet to relate the variables to the clauses. If the j^{th} literal of clause c_i is x_m, then we use an A widget to connect edge $(b_{ij}, b_{i,j+1})$ with edge e_m. If the j^{th} literal of clause c_i is $\neg x_m$, then we instead put an A widget between edge $(b_{ij}, b_{i,j+1})$ and edge $\bar{e}_m$. In fig. 5.48, for example, because clause C_2 is $(x_1 \vee \neg x_2 \vee x_3)$, we place three A widgets as follows –

(i) between $(b_{2,1}, b_{2,2})$ and e_1

(ii) between $(b_{2,2}, b_{2,3})$ and $\bar{e}_2$, and

(iii) between $(b_{2,3}, b_{2,4})$ and e_3.

Note that connecting two edges by means of A widgets actually entails replacing each edge by the five edges in the top or bottom of fig. 5.46 (a) and, of course, adding the connections that pass through the z vertices as well. A given literal l_m may appear in several clauses (for example, $\neg x_3$ in fig. 5.48), and thus an edge e_m or $\bar{e}_m$ may be influenced by several A widgets (for example, edge $\bar{e}_3$). In this case, we connect the A widgets in series, as shown in fig. 5.49, effectively replacing edge e_m or $\bar{e}_m$ by a series of edges.



(a) A Portion of Fig. 5.47 (b) The actual Subgraph Constructed

Fig. 5.49

We claim that formula ϕ is satisfiable if and only if graph G contains a Hamiltonian cycle. We first suppose that G has a Hamiltonian cycle h and show that ϕ is satisfiable. Cycle h must take a particular form –

(i) First, it traverses edge $(b_{1,1}, x'_1)$ to go from the top left to the top right.

(ii) It then follows all of the x'_m and x''_m vertices from top to bottom, choosing either edge e_m or edge $\bar{e}_m$, but not both.

(iii) It next traverses edge $(b_{k,4}, x''_n)$ to get back to the left side.

(iv) Finally, it traverses the B widgets from bottom to top on the left.

Given the Hamiltonian cycle h, we define a truth assignment for ϕ as follows. If edge e_m belongs to h, then we set $x_m = 1$. Otherwise, edge $\bar{e}_m$ belongs to h, and we set $x_m = 0$.

We claim that this assignment satisfies ϕ. Consider a clause C_i and the corresponding B widget in G. Each edge $(b_{i,j}, b_{i,j+1})$ is connected by an A widget to either edge e_m or edge $\bar{e}_m$, depending on whether x_m or $\neg x_m$ is the j^{th} literal in the clause. The edge $(b_{i,j}, b_{i,j+1})$ is traversed by h if and only if the corresponding literal is 0. Since each of the three edges $(b_{i,1}, b_{i,2})$, $(b_{i,2}, b_{i,3})$, $(b_{i,3}, b_{i,4})$ in clause C_i is also in a B widget, all three cannot be traversed by the Hamiltonian cycle h. One of the three edges, therefore, must have a corresponding literal whose assigned value is 1, and clause C_i is satisfied. This properly holds for each clause C_i, i = 1, 2, ..., k, and thus formula ϕ is satisfied.

Conversely, let us suppose that formula ϕ is satisfied by some truth assignment. By following the rules from above, we can construct a Hamiltonian cycle for graph G – traverse edge e_m if $x_m = 1$, traverse edge $\bar{e}_m$ if $x_m = 0$, and traverse edge $(b_{i,j}, b_{i,j+1})$ if and only if the j^{th} literal of clause C_j is 0 under the assignment. These rules can indeed be followed, since we assume that s is a satisfying assignment for formula ϕ.

Finally, we note that graph G can be constructed in polynomial time. It contains one B widget for each of the k clauses in ϕ. There is one A widget for each instance of each literal in ϕ, and so there are 3 k A widgets. Since the A and B widgets are of fixed size, the graph G has O(K) vertices and edges and is easily constructed in polynomial time. Thus, we have provided a polynomial-time reduction from 3-CNF-SAT to HAM-CYCLE.

Q.49. What is vertex cover ? Show that it is NP-complete.
(R.G.P.V., Dec. 2012)

Ans. **Vertex Cover** – Let G = (V, E) be an (undirected) graph with set of vertices V and edges E. A subset $A \subseteq V$ is said to be vertex cover of G if for every edge (v, w) in E, at least one of v or w is in A.

The vertex cover problem is – given a graph G and integer $\bar{k}$, does G have a vertex cover of size k or less ?

To represent this problem as a language L_{vc}, consisting of strings of the form– k in binary, followed by a marker, followed by the list of vertices, where v_i is represented by v followed by i in binary, and a list of edges, where (v_i, v_j) is represented by the codes for v_i and v_j surrounded by parentheses. L_{vc} consists of all such strings representing k and G, such that G has a vertex cover of size k or less.

Proof – Let A subset of k vertices and check that it covers all edges. This may be done in time proportional to the square of the length of the problem representation. L_{vc} is shown to be NP-complete by reducing 3-CNF satisfiability to L_{vc}.

Let $F = F_1 \wedge F_2 \wedge \ldots\ldots\ldots \wedge F_q$ be an expression in 3-CNF, where each F_i is a clause of the form $(\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$ each α_{ij} being a literal. We construct an undirected graph G = (V, E) whose vertices are pairs if integers (i, j), $1 \le i \le 2$, $1 \le j \le 3$. The vertex (i, j) represents the jth literal of the ith clause. The edges of the graph are

(i) [(i, j), (i, k)] provided $j \neq k$ and

(ii) [(i, j), (k, l)] if $\alpha_{ij} = \neg \alpha_{kl}$.

Each pair of vertices corresponding to the same clauses are connected by an edge in (i). Each pair of vertices corresponding to a literal and its complement are connected by an edge in (ii).

G has been constructed so that it has a vertex cover of size 2q if and only if F is satisfiable. Assume F is satisfiable and fix an assignment satisfying F. Each clause must have a literal whose value is 1. Select one such literal for each clause. Delete the q vertices corresponding to these literal from V. The remaining vertices form a vertex cover of size 2q. Clearly for each i, only one vertex of the form (i, j) is missing from the cover, and hence each edge in (i) is incident upon at least one vertex in the cover. Since edges in (ii) are incident upon two vertices corresponding to some literal and its complement, and since we would not have deleted both a literal and its complement, one or the other of these vertices is in the cover. Thus we indeed have a cover of size 2q.

Conversely, assume we have a vertex cover of size 2q, for each i the cover must contain all but one vertex of the form (i, j) for if two such vertices were missing, an edge [(i, j), (i, k)] would not be incident upon any vertex in the cover. There can be no conflict because two vertices not in the cover cannot correspond to a literal and its complement, else there would be an edge

in group (ii) not incident upon any vertex of the cover. For this assignment F has value 1. Thus F is satisfiable. We can essentially use the variables names in the formula F as the vertices of G, appending two bits for the j-component in vertex (i, j). Edges of type (i) are generated directly from the clauses, while those of type (ii) require two counters to consider all pairs of literals. Thus we conclude that L_{vc} is NP-complete.

★★★

RGPV

B.E. (Fourth Semester) EXAMINATION, Dec. 2012
ANALYSIS AND DESIGN OF ALGORITHM
(CS/IT-404)

Note : **(i)** Attempt *one* question from each Unit.
(ii) All questions carry equal marks.

Unit-I

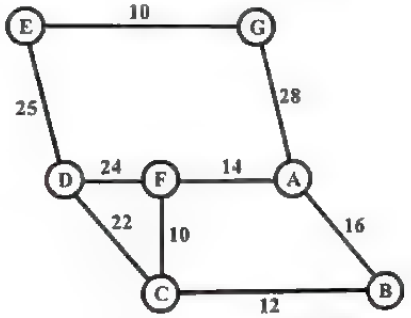
1. **(a)** What is the significance of various asymptotic notations ? Explain by giving examples. **(See Unit-I, Page 11, Q.15)**
   (b) Write short note on Strassen's Matrix multiplication. Compare it with conventional Divide and conquer technique of matrix multiplication. **(See Unit-I, Page 50, Q.61)**

Or

2. **(a)** Explain how time complexity of an algorithm can be calculated ? Give suitable example. **(See Unit-I, Page 7, Q.9)**
   (b) What is divide and conquer strategy ? Write and explain the algorithm of Divide and conquer. **(See Unit-I, Page 31, Q.32)**

Unit-II

3. **(a)** What is greedy strategy ? Write prims minimum cost spanning tree algorithm. Determine MST for the following graph.



(See Unit-II, Page 95, Prob.11)

   (b) Find an optimal solution to the knapsack instance n = 7, m = 15. $(P_1, P_2 \ldots P_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(W_1, W_2, \ldots W_7) = (2, 3, 5, 7, 1, 4, 1)$. **(See Unit-II, Page 100, Prob.15)**

Or

4. **(a)** Find the optimal merge pattern for following data :
   28, 32, 12, 5, 84, 53, 91, 35, 3, 11
   (See Unit-II, Page 85, Prob.2)
   (b) Obtain a set of optimal Huffman code for the messages $(m_1, m_2, \ldots, m_7)$ with relative frequencies $(q_1, \ldots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes. **(See Unit-II, Page 88, Prob.5)**

Unit-III

5. **(a)** What are the key ingredients that an optimization problem must have

for dynamic programming to be applicable ? Explain them.
(See Unit-III, Page 112, Q.7)

(b) Give Floyd Warshall Algorithm. What is its running time ?
(See Unit-III, Page 124, Q.21)

Or

6. (a) Determine an LCS of <1, 0, 0, 1 0, 1, 0, 1> and <0, 1, 0, 1, 1, 0, 1, 1, 0>. How dynamic programming can be applied to LCS ?
(See Unit-III, Page 135, Prob.12)

(b) Define all-pairs shortest path problem. Discuss solution of this problem based on Dynamic programming. (See Unit-III, Page 122, Q.20)

Unit-IV

7. (a) Explain backtracking. Write an algorithm to estimate the efficiency of backtracking. (See Unit-IV, Page 137, Q.1)

(b) What is graph colouring program ? Give an algorithm for 3 colouring of a graph. (See Unit-IV, Page 154, Q.20)

Or

8. (a) Explain how branch and bound method can be used to solve any problem ? What is least cost search ? (See Unit-IV, Page 159, Q.25)

(b) Explain the following with suitable examples.
(i) Hamiltonian cycle (See Unit-IV, Page 148, Q.15)
(ii) Parallel Algorithms (See Unit-IV, Page 182, Q.33)

Unit-V

9. (a) Define B-tree-Insert the entires below, in the order stated, into an initially empty B-tree of order 4.
1, 5, 6, 2, 8, 11, 13, 18, 20, 7, 9 (See Unit-V, Page 218, Prob.11)

(b) What is binary search tree ? Give a comparison of binary search tree and Heap. (See Unit-V, Page 185, Q.2)

Or

10. (a) What do you understand by NP-hard and NP-complete classes ? Explain. (See Unit-V, Page 240, Q.46)

(b) What is vertex cover ? Show that it is NP-complete.
(See Unit-V, Page 246, Q.49)

B.E. (Fourth Semester) EXAMINATION, June 2013
ANALYSIS AND DESIGN OF ALGORITHM
(CS/IT-404)

Note : Attempt one question from each unit, including sub parts. All questions carry equal marks.

Unit-I

1. (a) Describe the methods of analyzing an algorithm. What do you mean by best case, average case and worst case time complexity of an algorithm. (See Unit-I, Page 6, Q.8)

(b) Explain divide and conquer technique. Design a recursive algorithm

for binary search.
(See Unit-I, Page 36, Q.42)

Or

2. (a) Explain heap sort algorithm with example. (See Unit-I, Page 23, Q.30)

(b) Solve the recurrence relation –
$T(n) = 3(n/4) + n$
(See Unit-I, Page 53, Prob.6)

Unit-II

3. (a) Obtain a set of optimal Huffman codes for the seven messages $(M_1,.....,M_7)$ with relative frequencies $(q_1,.....,q_7) = (4, 5, 7, 8, 10, 22, 15)$. Draw the decode tree for this set of codes.
(See Unit-II, Page 89, Prob.6)

(b) Write and explain single source shortest path algorithm with example.
(See Unit-II, Page 81, Q.24)

Or

4. (a) Consider the Knapsack instance $n = 3$, $(W_1, W_2, W_3) = (2, 3, 4)$ and $(P_1, P_2, P_3) = (1, 2, 5)$ and $m = 5$. Find the optimal solution.
(See Unit-II, Page 101, Prob.16)

(b) There are 5 jobs whose profits $(P_1,.....,P_5) = (20, 15, 10, 1, 6)$ and deadlines (2, 2, 1, 3, 3). Find the optimal solution that minimizes profit on scheduling these jobs. Discuss its algorithm too. (See Unit-II, Page 105, Q.19)

Unit-III

5. (a) Write Floyd-Warshall algorithm to solve all pair shortest path problem. Also write its complexity. (See Unit-III, Page 124, Q.21)

(b) Show that greedy strategy will not work for 0-1 Knapsack problem. Give a dynamic programming based solution for this problem.
(See Unit-III, Page 115, Q.12)

Or

6. (a) What is multistage graph problem ? Discuss its solution based on dynamic programming approach. Give a suitable algorithm and find its computing time ? (See Unit-III, Page 118, Q.16)

(b) Explain dynamic programming concept with example.
(See Unit-III, Page 110, Q.3)

Unit-IV

7. (a) Explain backtracking technique for designing an algorithm.
(See Unit-IV, Page 139, Q.4)

(b) What is Hamiltonian cycle ? Write an algorithm to find all Hamiltonian cycle in graph ? (See Unit-IV, Page 148, Q.16)

Or

8. (a) What is branch and bound technique ? How travelling sales person problem can be solved using this technique ? (See Unit-IV, Page 161, Q.28)

(b) What is graph coloring problem ? Give algorithm to solve this problem?
(See Unit-IV, Page 154, Q.20)

Unit-V

9. (a) Create a B-tree for the following list of elements –
{86, 50, 40, 3, 94, 10, 70, 90, 110, 113, 116}
Given minimization factor t = 3, minimum degree = 2 and maximum

degree = 5. (See Unit-V, Page 219, Prob.12)

(b) Show that the travelling sales man problem is NP-complete. (See Unit-V, Page 241, Q.47)

dOr

10.(a) Write DFS and BFS algorithms and also analyses the running time of algorithm. (See Unit-V, Page 236, Q.35)

(b) Create an AVL tree for the following. List of elements by inserting in empty AVL tree (Write step by step insertion).

RGPV

B.E. (Fourth Semester) EXAMINATION, Dec. 2013
ANALYSIS AND DESIGN OF ALGORITHM
(CS/IT-404)

Note : Question paper is divided into five units. Attempt one question from each unit. All questions carry equal marks.

Unit-I

1. (a) Write in brief about the significance of the following notations – 7
(i) 'O' (ii) 'Ω' (iii) 'θ' (iv) 'o' (v) 'ω'
(See Unit-I, Page 16, Q.23)

(b) How time complexity of an algorithm is calculated ? Explain with suitable example. (See Unit-I, Page 7, Q.9) 7

Or

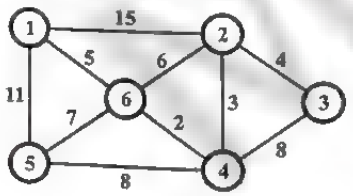
2. (a) Compare Heap and binary search tree. Construct a heap and a binary search tree of the following data –
44, 22, 77, 11, 66, 55, 33, 88, 99 (See Unit-V, Page 207, Prob.1)

(b) Explain the divide and conquer technique. Write an algorithm to find the maximum and minimum element in an array using this technique. 7
(See Unit-I, Page 32, Q.33)

UNIT-II

3. (a) What is greedy strategy ? Explain optimal merge pattern in brief and find an optimal binary merge tree (pattern) for files whose lengths are 28, 32, 12, 15, 84, 53, 91, 35, 5 and 11. (See Unit-II, Page 85, Prob.3) 7

(b) What is minimum spanning tree ? Using Prim's algorithm find minimum spanning tree of the following graph. 7



(See Unit-II, Page 94, Prob.10)

Or

4. (a) What is Knapsack problem, find the optimal solution to the Knapsack

instance n = 3, m = 20, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$. (See Unit-II, Page 100, Prob.14) 7

(b) Write an algorithm for job sequencing problem with deadline. Also discuss its complexity. (See Unit-II, Page 79, Q.22) 7

UNIT-III

5. (a) What is dynamic programming ? Compare it with greedy method. 7
(See Unit-III, Page 112, Q.6)

(b) What is LCS problem ? Write an algorithm to solve longest common sequence problem using dynamic programming approach. 7
(See Unit-III, Page 126, Q.22)

Or

6. (a) Explain Floyd Warshall algorithm with the help of an example. 7
(See Unit-III, Page 124, Q.21)

(b) What is reliability design problem ? Explain. (See Unit-III, Page 120, Q.17) 7

UNIT-IV

7. (a) Explain Backtracking. How backtracking algorithm can be used to solve 8 queens problem ? (See Unit-IV, Page 143, Q.10) 7

(b) What are hamiltonian cycles ? Write an algorithm to find all Hamiltonian cycles in a given graph. (See Unit-IV, Page 148, Q.16) 7

Or

8. (a) What is branch and bound method, using branch and bound method generate a state space tree for the following cost matrix. 7

$C_{ij} =$

	A	B	C	D
A	∞	12	7	4
B	10	∞	13	9
C	3	8	∞	11
D	5	6	10	∞

(See Unit-IV, Page 168, Q.30)

(b) Write about parallel algorithms with example. 7
(See Unit-IV, Page 182, Q.33)

UNIT-V

9. (a) Define height balance tree. Construct a height balance tree starting with empty tree on the following data –
Dec, Jan, Apr, Mar, Jul, Aug, Oct, Feb, Nov, May, Jun. 7
(See Unit-V, Page 215, Prob.8)

(b) Create a B-tree of order 5 for the following data items –
D, H, K, J, B, P, Q, E, A, S, W, T, C, L, N, Y, M.
(See Unit-V, Page 220, Prob.13)

Or

10. Explain the following with examples. 7 × 2
(a) 2-3 trees (See Unit-V, Page 200, Q.12)
(b) NP-Completeness. (See Unit-V, Page 239, Q.45)

RGPV

B.E. (Fourth Semester) EXAMINATION, June 2014
ANALYSIS AND DESIGN OF ALGORITHM
(CS/IT-404)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.
(ii) All parts of each question are to be attempted at one place.
(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.
(iv) Except numericals, Derivation, Design and Drawing etc.

1. (a) What are different asymptotic notations used ? Explain. 2
(See Unit-I, Page 11, Q.15)

(b) Explain the strassen's multiplication technique. 2
(See Unit-I, Page 50, Q.61)

(c) Explain how to apply the divide and conquer strategy for sorting the elements using quick sort ? (See Unit-I, Page 45, Q.54) 3

(d) Following nodes are inserted in empty tree to form minimum heap. With neat sketches show how insertion will be done 8, 7, 11, 6, 2, 1, 5, 12. (See Unit-I, Page 24, Prob.2) 7

Or

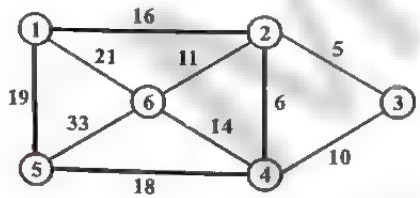
Sort the given list using merge sort 70, 80, 40, 50, 60, 12, 35, 95, 10. 7
(See Unit-I, Page 58, Prob.11)

2. (a) Explain optimal merge (pattern) in brief. (See Unit-II, Page 66, Q.5) 2

(b) Write the general characteristics of greedy algorithm. 2
(See Unit-II, Page 66, Q.4)

(c) Give an algorithm for computing minimum spanning tree. 3
(See Unit-II, Page 72, Q.12)

(d) Find minimum spanning tree using Prim's algorithm for graph given below. 7



(See Unit-II, Page 94, Prob.9)

Or

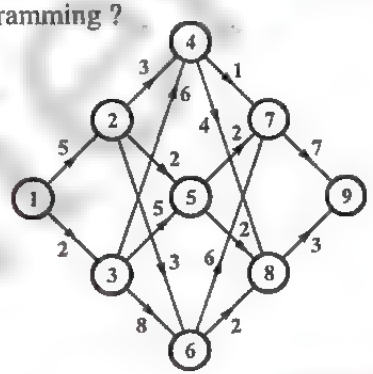
Explain how job sequencing with deadline can be solved using greedy approach. (See Unit-II, Page 79, Q.21) 7

3. (a) What is the principle of optimality ? Explain with example. 2
(See Unit-III, Page 111, Q.5)

(b) Define multistage graph problem with the help of suitable algorithm. 2
(See Unit-III, Page 118, Q.16)

(c) Find optimal solution for 0/1 Knapsack problem (w_1,w_2,w_3,w_4) = (10, 15, 6, 9); (P_1, P_2, P_3, P_4) = (2, 5, 8, 1) and M = 30. 3
(See Unit-III, Page 129, Prob.4)

(d) Find a minimum cost path from 'S' to 't' in multistage graph using dynamic programming ? 7



(See Unit-III, Page 130, Prob.6)

Or

Explain Floyd-Warshall algorithm with suitable example. 7
(See Unit-III, Page 124, Q.21)

4. (a) Explain the concept of backtracking. (See Unit-IV, Page 139, Q.4) 2

(b) Write a pseudo algorithm for graph coloring problem. 2
(See Unit-IV, Page 153, Q.19)

(c) What is Hamiltonian cylce ? Explain how it can be solved using backtracking algorithm. (See Unit-IV, Page 148, Q.16) 3

(d) Draw the portion of state space tree generated by LC branch and bound for the following Knapsack instance n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18); (w_1, w_2, w_3, w_4) = (2, 4, 6, 9) and M = 15. 7
(See Unit-IV, Page 178, Prob.7)

Or

Consider the travelling salesman on instance defined by cost matrix –

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

Obtain the reduced cost matrix and solve it. 7
(See Unit-IV, Page 168, Prob.3)

5. (a) Explain 2-3 trees with the help of suitable example. 2
(See Unit-V, Page 200, Q.12)

(b) What is AVL tree ? Discuss its properties. (See Unit-V, Page 191, Q.6) 7

(c) What is P, NP class problems ? Explain the concept with suitable example. (See Unit-V, Page 238, Q.42) 3

(d) Insert the elements in the order shown to build them into an AVL tree. Also determine the complexity of this procedure 1, 26, 2, 25, 3, 24, 4, 23, 5, 22, 6. (See Unit-V, Page 215, Prob.9) 7

Or

Create a B-tree for the following list of elements L = {86, 50, 40, 3, 94, 10, 70, 90, 110, 113, 116} given minimization factor t = 3, minimum degree = 2 and maximum degree = 5. (See Unit-V, Page 219, Prob.12) 7

RGPV

B.E. (Fourth Semester) EXAMINATION, Dec. 2014
ANALYSIS AND DESIGN OF ALGORITHM
(CS/IT-404)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.
(ii) All parts of each question are to be attempted at one place.
(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.
(iv) Except numericals, Derivation, Design and Drawing etc.

1. (a) What do you mean by performance analysis of an algorithm ? Explain. (See Unit-I, Page 5, Q.5)

(b) What are the different asymptotic notations used ? Explain. (See Unit-I, Page 11, Q.15)

(c) Explain any one application that can be solved by divide and conquer. (See Unit-I, Page 34, Q.35)

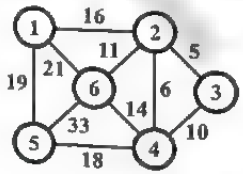
(d) Write down Strassen's algorithm for multiplication. (See Unit-I, Page 50, Q.61)

Or

How recursive algorithms are analyzed ? Analyze the execution time of recursive algorithm for tower of Hanoi; problem. (See Unit-I, Page 9, Q.11)

2. (a) What do you mean by feasible solution ? (See Unit-II, Page 65, Q.2)
(b) What is minimum spanning tree ? (See Unit-II, Page 71, Q.10)
(c) Write the general characteristics of Greedy algorithm. (See Unit-II, Page 66, Q.4)

(d) Find the shortest path from vertex 1 to vertex 3 in the following weighted graph using Dijkstra's greedy algorithm.



(See Unit-II, Page 107, Prob.23)

Or

Find the optimal binary merge tree (pattern) for ten files whose length are 28, 32, 12, 5, 84, 53, 91, 35, 3 and 11. Also find its weighted external path length. (See Unit-II, Page 84, Prob.1)

3. (a) What is principle of optimality ? Explain with example. (See Unit-III, Page 111, Q.5)

(b) Write the characteristics of dynamic programming. (See Unit-III, Page 111, Q.4)

(c) Give the commonly used designing steps for dynamic programming algorithm. (See Unit-III, Page 113, Q.8)

(d) Consider the Knapsack instance with 5 objects and a capacity m = 11, profits p = (5, 4, 7, 2, 3) and weights w = (4, 3, 6, 2, 2). Solve it using dynamic programming approach. (See Unit-III, Page 128, Prob.3)

Or

What is multistage graph problem ?Discuss its solution based on dynamic programming approach. Give a suitable algorithm and find its computing time. (See Unit-III, Page 118, Q.16)

4. (a) Explain the use of bounding function. (See Unit-IV, Page 157, Q.23)

(b) Explain how to solve sum of subset problem ? (See Unit-IV, Page 160, Q.26)

(c) What is Hamiltonian cycle ? Explain how it can be solved using backtracking algorithm. (See Unit-IV, Page 148, Q.16)

(d) Draw the portion of state space tree generated by LC branch and bound for the following Knapsack instance n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18); (w_1, w_2, w_3, w_4) = (2, 4, 6, 9) and M = 15. (See Unit-IV, Page 178, Prob.7)

Or

Consider the travelling salesman on instance defined by the cost matrix–

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

Obtain the reduced cost matrix and solve it. (See Unit-IV, Page 168, Prob.3)

5. (a) Explain binary search tree. List out its properties. (See Unit-V, Page 185, Q.1)

(b) Explain 2-3 trees with the help of suitable example. (See Unit-V, Page 200, Q.12)

(c) What is P, NP class problems ? Explain the concept with suitable example. (See Unit-V, Page 238, Q.42)

(d) Obtain height balanced trees starting with empty tree on the following set of instructions –
Dec, Jan, Apr, Mar, Jul, Aug, Oct, Feb, Nov, May, June. (See Unit-V, Page 213, Prob.7)

Or

Construct an AVL tree for the following list {5, 6, 8, 3, 2, 4, 7} by inserting the elements successively, starting with empty tree.
(See Unit-V, Page 210, Prob.4)

B.E. (Fourth Semester) EXAMINATION, June 2015
ANALYSIS AND DESIGN OF ALGORITHM
(CS/IT-404)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.
(ii) All parts of each question are to be attempted at one place.
(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.
(iv) Except numericals, Derivation, Design and Drawing etc.

Unit-I

1. (a) Define Ω notation. (See Unit-I, Page 14, Q.17)
   (b) What is the use of asymptotic notations ? (See Unit-I, Page 11, Q.14)
   (c) Define binary tree. (See Unit-V, Page 185, Q.1)
   (d) How divide and conquer technique can be applied to binary trees? Also write algorithm for divide and conquer. (See Unit-I, Page 34, Q.36)

   Or

   Define and explain merge sort algorithm. (See Unit-I, Page 44, Q.50)

Unit-II

2. (a) Explain about greedy technique. (See Unit-II, Page 65, Q.1)
   (b) Define the external path length. (See Unit-II, Page 69, Q.8)
   (c) Define minimum spanning tree. (See Unit-II, Page 71, Q.10)
   (d) Define Kruskal's algorithm. Also write down the steps for Kruskal's algorithm in detail. (See Unit-II, Page 74, Q.13)

   Or

   Define preorder, inorder and postorder traversal. Also explain in detail all the traversals. (See Unit-V, Page 229, Q.25)

Unit-III

3. (a) Explain principle of optimality. (See Unit-III, Page 111, Q.5)
   (b) Define dynamic programming. (See Unit-III, Page 109, Q.1)
   (c) Define binomial coefficient. (See Unit-III, Page 113, Q.9)
   (d) Explain Warshall's algorithm. (See Unit-III, Page 124, Q.21)

   Or

   Find an optimal solution to the following Knapsack problem –
   Number of objects n = 3
   Knapsack capacity m = 20
   Profits $(p_1, p_2, p_3) = (25, 24, 15)$
   Weights $(w_1, w_2, w_3) = (18, 15, 10)$.
   (See Unit-II, Page 98, Prob.13)

Unit-IV

4. (a) Explain state space tree. (See Unit-IV, Page 140, Q.5)
   (b) Explain n-queens problem. (See Unit-IV, Page 146, Q.11)
   (c) Explain graph coloring problem. (See Unit-IV, Page 151, Q.18)
   (d) How can travelling salesperson problem be solved ? (See Unit-IV, Page 162, Q.28)

   Or

   Explain backtracking in detail. Also write algorithm for recursive backtracking algorithm. (See Unit-IV, Page 139, Q.4)

Unit-V

5. (a) Explain class P problem. (See Unit-V, Page 238, Q.40)
   (b) Explain undecidable problem. (See Unit-V, Page 237, Q.37)
   (c) Explain NP-hard problems. (See Unit-V, Page 239, Q.43)
   (d) List out the techniques for traversals in graph. Also explain each in detail with its procedure. (See Unit-V, Page 236, Q.32)

   Or

   Discuss the relationship between P, NP, NP-complete and NP-hard problems. (See Unit-V, Page 239, Q.44)

B.E. (Fourth Semester) EXAMINATION, Dec. 2015
ANALYSIS AND DESIGN OF ALGORITHMS
(CS/IT-404)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.
(ii) All parts of each question are to be attempted at one place.
(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.
(iv) Except numericals, Derivation, Design and Drawing etc.

1. (a) Show that solution of $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$ is $O(\log n)$.
   (b) What are the factors which affect the running time of an algorithm ? (See Unit-I, Page 8, Q.10)
   (c) Illustrate Heap sort on the array;
   A = [5, 8, 3, 9, 2, 10, 1, 40] (See Unit-I, Page 28, Prob.4)
   (d) Sort the following list using quick sort technique and argue upon its running time
   A = [5, 7, 9, 4, 10, 2, 8, 1] (See Unit-I, Page 59, Prob.12)

   Or

   Show how the following matrices would be multiplied using Strassen's algorithm.

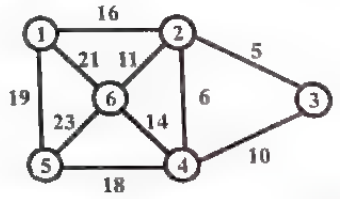
2. (a) Explain the concept behind Greedy strategy. (See Unit-II, Page 65, Q.1)
   (b) Write the basic difference between Prim's algorithm and Kruskal's algorithm. (See Unit-II, Page 76, Q.15)

(c) Consider n = 7, m = 15, $(P_1, P_2, \ldots P_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w_1, w_2, \ldots w_7) = (2, 3, 5, 7, 1, 4, 1)$. Obtain the optimal solution for this Knapsack instance. (See Unit-II, Page 100, Prob.15)

(d) Find on optimal merge pattern for 11 files whose length are 28, 32, 12, 5, 84, 5, 3, 9, 35, 3, 11. Write and explain the algoritm used and determine its complexity. (See Unit-II, Page 87, Prob.4)

Or

Write Dijkstra's algorithm to find the shortest path between two given vertices. Using this algorithm find shortest path from vertex 1 to vertex 3 in the following weighted graph.



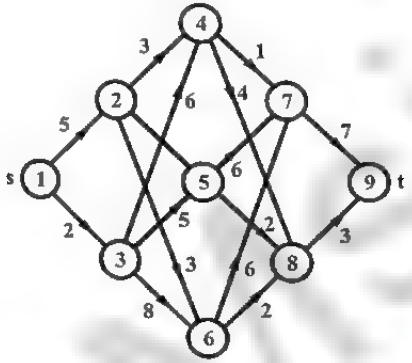
(See Unit-II, Page 108, Prob.24)

3. (a) What is principle of optimality ? Explain with suitable example. (See Unit-III, Page 111, Q.5)

(b) Write short note on dynamic programming. (See Unit-III, Page 109, Q.1)

(c) Explain how a reliability design can be obtained using dynamic programming. (See Unit-III, Page 120, Q.17)

(d) Find minimum cost path from 'S' to 't' in multistage graph using dynamic programming ?



(See Unit-III, Page 131, Prob.7)

Or

Define how Knapsack problem is solved using dynamic programming approach.

Consider

n = 3, $(w_1, w_2, w_3) = (2, 3, 3)$ and $(p_1, p_2, p_3) = (1, 2, 4)$, m = 6. Find optimal solution for given data. (See Unit-III, Page 127, Prob.2)

4. (a) Write a Pseudo algorithm for graph coloring problem. (See Unit-IV, Page 153, Q.19)

(b) Explain the term lower bound with suitable example. (See Unit-IV, Page 180, Q.31)

(c) What is Hamiltonian cycle ? Explain how it can be solved using backtracking algorithm. (See Unit-IV, Page 148, Q.16)

(d) Consider the following travelling salesman on instance defined by cost matrix.

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

obtain reduced cost matrix and solve it. (See Unit-IV, Page 168, Prob.3)

Or

Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1). (See Unit-IV, Page 147, Q.12)

5. (a) Explain the function for deletion of a node from binary search tree. (See Unit-V, Page 189, Q.5)

(b) Explain 2-3 trees with the help of suitable example. (See Unit-V, Page 200, Q.12)

(c) What is BFS and DFS ? Explain with suitable example with respect to tree. (See Unit-V, Page 233, Q.31)

(d) Insert the elements in the order shown below to build into an AVL tree. Also determine the complexity of this procedure 1, 26, 2, 25, 3, 24, 4, 23, 5, 22, 6. (See Unit-V, Page 215, Prob.9)

Or

Discuss the relationship between class P, NP, NP complete and NP hard problems with example of each class. (See Unit-V, Page 239, Q.44)

RGPV

B.E. (Fourth Semester) EXAMINATION, June 2016

ANALYSIS AND DESIGN OF ALGORITHMS

(CS/IT-404)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.

(ii) All parts of each question are to be attempted at one place.

(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.

(iv) Except numericals, Derivation, Design and Drawing etc.

1. (a) Discuss in which condition binary search is better than linear search. (See Unit-I, Page 37, Q.43)

(b) Give the definition of Big "oh" with example. (See Unit-I, Page 14, Q.16)

(c) Define algorithm. Discuss how to analyse algorithms. (See Unit-I, Page 5, Q.7)

(d) Write the procedure of merge sort and sort the given array of 8 elements step-by-step using merge sort 35, 18, 7, 12, 5, 23, 16, 3. (See Unit-I, Page 57, Prob.9)

Or

Discuss Strassen's algorithm for matrix multiplication with example. (See Unit-I, Page 50, Q.61)

2. (a) What is greedy approach ? (See Unit-II, Page 69, Q.1)

(b) How two way merge pattern can be represented by binary merge tree ? (See Unit-II, Page 67, Q.6)

(c) Discuss job sequencing problem by an example. (See Unit-II, Page 79, Q.20)

(d) Find an optimal solution to the knapsack instance n = 3, m = 20, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = = (18, 15, 10)$. (See Unit-II, Page 98, Prob.13)

Or

Given E is the set of edges in graph G. G has n vertices. Cost [u, v] is the cost of edge (u, v). T is the set of edges in the minimum-cost spanning tree. Write the pseudocode for Kruskal Algorithm by considering parameters mentioned above. (See Unit-II, Page 74, Q.13)

3. (a) Write short note on dynamic programming. (See Unit-III, Page 109, Q.1)

(b) How reliability of a system is calculated ? (See Unit-III, Page 120, Q.17)

(c) Discuss 0/1 Knapsack problem. (See Unit-III, Page 114, Q.11)

(d) Explain multistage graphs. Discuss its applications. (See Unit-III, Page 117, Q.15)

Or

Discuss Floyd-Warshall algorithm. Write its pseudocode. (See Unit-III, Page 124, Q.21)

4. (a) Define branch and bound method. (See Unit-IV, Page 157, Q.23)

(b) Give the definition of Hamiltonian cycle. (See Unit-IV, Page 147, Q.13)

(c) Discuss graph colouring for complete graphs.

(d) What do you understand by travelling salesman problem ? Discuss with suitable example. (See Unit-IV, Page 163, Q.29)

Or

Explain eight queen's problem and apply backtracking to solve this problem. (See Unit-IV, Page 143, Q.10)

5. (a) Write the rules to construct binary search tree. (See Unit-V, Page 185, Q.1)

(b) Compare DFS and BFS. (See Unit-V, Page 231, Q.29)

(c) Discuss polynomial time and non-polynomial time algorithms. (See Unit-V, Page 237, Q.38)

(d) Define height balanced tree. Explain all the rotations perform to balance the tree with example. (See Unit-V, Page 196, Q.9)

Or

The post order traversal of a binary tree T is DFEBGLJKHCA and inorder traversal of T is DBFEAGCLJHK. Construct the binary tree T and also writes the steps to construct the binary tree in postorder-inorder combination. (See Unit-V, Page 208, Prob.2)

RGPV

B.E. (Fourth Semester) EXAMINATION, Dec. 2016

ANALYSIS AND DESIGN OF ALGORITHMS

(CS/IT-404)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.
(ii) All parts of each question are to be attempted at one place.
(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (Max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.
(iv) Except numericals, Derivation, Design and Drawing etc.

Unit-I

1. (a) What is the data structures used to perform recursion ? (See Unit-I, Page 33, Q.34)

(b) What is the purpose of Strassen's matrix multiplication ? (See Unit-I, Page 50, Q.60)

(c) Why do we use asymptotic notations in the study of algorithms ? Briefly describe the commonly used asymptotic notation. (See Unit-I, Page 14, Q.19)

(d) Apply quick sort algorithm for the following array and sort the element (Take first element of the list as the pivot element).
25, 56, 47, 35, 10, 90, 82, 31.
Also discuss complexity of algorithm. (See Unit-I, Page 62, Prob.15)

Or

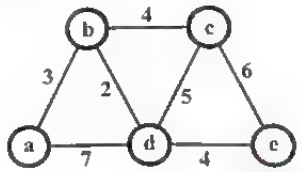
Sort the following list using heap sort –
66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65.
Also discuss the complexity of the heap sort. (See Unit-I, Page 25, Prob.3)

Unit-II

2. (a) What is greedy techniques ? Derive the equation for the optimal solutions. (See Unit-II, Page 66, Q.3)

(b) Tabulate the differences between Kruskal's and Prim's algorithm. (See Unit-II, Page 76, Q.15

(c) Solve the following instances of the single source shortest patl problem with vertex 'a' as the source.



(See Unit-II, Page 106, Prob.22)

(d) Construct a Huffman code for the following data –

Character	A	B	C	D	E
Probability	0.4	0.1	0.2	0.15	0.15

Decode the text whose ending 100010111001010 using the above huffman code. (See Unit-II, Page 91, Prob.7)

Or

Write a greedy algorithm for sequencing unit time jobs with deadlines and profits. Using this algorithm, find the optimal solution when n = 5.

Job	*Profit*	*Deadline*
P_1	20	2
P_2	15	2
P_3	10	1
P_4	5	3
P_5	1	3

(See Unit-II, Page 105, Prob.20)

Unit-III

3. (a) Explain the concept of dynamic programming. (See Unit-III, Page 109, Q.1)

(b) Give a dynamic programming solution for computing binomial coefficients. (See Unit-III, Page 114, Q.10)

(c) What is the concept of reliability design in dynamic programming. (See Unit-III, Page 120, Q.17)

(d) Using Floyd Warshall algorithm solve the all pair shortest path problem for the graph. Where weight matrix is given below –

$$\begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}$$

(See Unit-III, Page 134, Prob.11)

Or

Solve the following instance of 0/1 Knapsack problem using dynamic algorithm.

Item	1	2	3	4
Weight	4	7	5	3
Value	$40	$42	$25	$12

The capacity of Knapsack W is 10. (See Unit-III, Page 126, Prob.1)

Unit-IV

4. (a) Define hamiltonian cycle with example. (See Unit-IV, Page 148, Q.15)

(b) Explain graph coloring problem with their complexity. (See Unit-IV, Page 155, Q.21)

(c) What is backtracking ? Find a solution to the 4-Queen problem using backtracking strategy. (See Unit-IV, Page 142, Q.8)

(d) Solve the following instance of the Knapsack problem by the branch and bound algorithm.

Item	*Weight*	*Value*	$\frac{Value}{Weight}$
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

The Knapsack capacity W is 10. (See Unit-IV, Page 177, Prob.6)

Or

List out parallel algorithm and compare them with different factors.

Unit-V

5. (a) How multiway search is different from binary search tree ? (See Unit-V, Page 203, Q.18)

(b) Explain the concept of height balanced trees with their operations. (See Unit-V, Page 200, Q.11)

(c) Construct a B-tree of order 3 for the following set of input data – 69, 19, 43, 16, 25, 40, 132, 100, 145, 7, 15, 18. (See Unit-V, Page 222, Prob.14)

(d) Write any two data structures that are suitable for representing a graph. Write an algorithm for depth first traversal of a graph using one of your two data structures. (See Unit-V, Page 232, Q.30)

Or

Write a short notes on the following (any three)

(i) 2-3 tree (See Unit-V, Page 200, Q.12)
(ii) NP completeness problem (See Unit-V, Page 239, Q.45)
(iii) Lower bound theory (See Unit-IV, Page 180, Q.32)
(iv) Merge sort. (See Unit-I, Page 37, Q.45)

RGPV

B.E. (Fourth Semester) EXAMINATION, June 2017

ANALYSIS AND DESIGN OF ALGORITHMS

(CS/IT-404)

Note : (i) Attempt any five questions.
(ii) All questions carry equal marks.

1. (a) What are the differences between Big-Oh (O), Omega (ω) and Theta (θ) notations ? (See Unit-I, Page 14, Q.18)
   (b) Is there any difference among algorithm, pseudocode and program ? Explain. (See Unit-I, Page 3, Q.2)
2. (a) Apply binary search to find 123 in a list –
   45, 96, 105, 121, 145, 192, 199, 205, 245, 275, 123, 850, 905.
   (See Unit-I, Page 54, Prob.7)
   (b) Sort the following list using quick sort –
   36, 95, 42, 12, 08, 66, 72, 55 (See Unit-I, Page 59, Prob.13)
3. (a) How divide and conquer technique can be applied to binary trees ? Also write algorithm for divide and conquer. (See Unit-I, Page 34, Q.36)
   (b) Explain strassen's matrix multiplication with the help of an example.
   (See Unit-I, Page 51, Q.61)
4. (a) What is spanning tree ? Write Kruskals algorithm with an example to find minimal spanning tree. (See Unit-II, Page 76, Q.14)
   (b) A Knapsack capacity is 100. The weights and values of five objects are as follows –

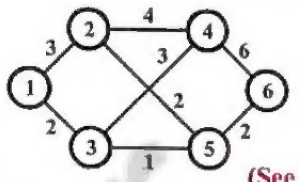
Weight W_i :	10	20	30	40	50
Value P_i :	20	30	66	20	60

   Solve the Knapsack problem using Greedy strategy and find the maximum profit that can be obtained. (See Unit-II, Page 102, Prob.17)
5. (a) Use the Floyd-warshall algorithm and find all pair shortest paths for the following adjacency weighted matrix.

$$\begin{bmatrix} 0 & 4 & \infty & 3 \\ \infty & 0 & 2 & 1 \\ 5 & 3 & 0 & \infty \\ 1 & \infty & 2 & 0 \end{bmatrix}$$

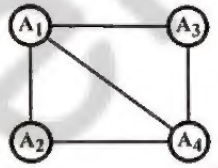
(See Unit-III, Page 133, Prob.10)

   (b) Solve the following multistage problem using both forward and backward reasoning.



(See Unit-III, Page 131, Prob.8)

6. (a) Colour the following graph using a vertex colouring algorithm. What is the minimum number of colours required ?



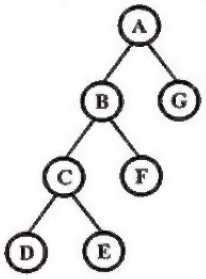
(See Unit-IV, Page 156, Prob.1)

   (b) Solve the TSP using branch and bound technique –

	A	B	C
A	∞	2	3
B	5	∞	3
C	2	4	∞

(See Unit-IV, Page 176, Prob.4)

7. (a) Show preorder, inorder and postorder for the following tree –



(See Unit-V, Page 230, Q.27)

   (b) What is a B-tree ? Write down the properties of a B-tree. Illustrate your answer with an example. (See Unit-V, Page 202, Q.16)
8. Write short notes –
   (a) Parallel algorithm (See Unit-IV, Page 182, Q.33)
   (b) NP completeness (See Unit-V, Page 239, Q.45)
   (c) Reliability design. (See Unit-III, Page 120, Q.17)

RGPV

CS-228 (CBCS)

B.E. (IV Semester) EXAMINATION, Dec. 2017

Choice Based Credit System (CBCS)

ANALYSIS & DESIGN OF ALGORITHM

Note : (i) Attempt any five questions.

(ii) Each question carries equal marks.

1. (a) What criteria are used during the analysis of the algorithm. (See Unit-I, Page 5, Q.6)

   (b) What is the significance of asymptotic notation ? (See Unit-I, Page 11, Q.13)

2. (a) Explain Strassen's matrix multiplication algorithm ? (See Unit-I, Page 51, Q.61)

   (b) Solve the following recurrence relation –

   $T(n)=3T\left(\frac{n}{4}\right)+n$ (See Unit-I, Page 53, Prob.6)

3. (a) Explain how to apply the divide and conquer strategy for sorting the elements using quick sort ? (See Unit-I, Page 45, Q.54)

   (b) Write and explain an algorithm to search an item in array using divide and conquer strategy with complexity $O(\log_2 n)$. (See Unit-I, Page 36, Q.40)

4. (a) Explain the greedy strategy. Write algorithm for Knapsack problem. (See Unit-II, Page 78, Q.18)

   (b) Find the optimal merge pattern for the following data – 28, 32, 12, 5, 84, 53, 91, 35, 3, 11. (See Unit-II, Page 85, Prob.2)

5. (a) Explain reliability design problem with suitable example. (See Unit-III, Page 120, Q.17)

   (b) Explain dynamic programming with example. (See Unit-III, Page 110, Q.3)

6. (a) Design a backtracking for graph-coloring problem. (See Unit-IV, Page 153, Q.19)

   (b) Explain and solve 4-queen's problem using backtracking. (See Unit-IV, Page 141, Q.7)

7. (a) A binary tree T has 9 nodes the inorder and preorder traversal of T yield the following sequency of nodes –
   Inorder : E A C K F H D B G
   Preorder : F A E K C D H G B
   Draw the tree T. (See Unit-V, Page 209, Prob.3)

   (b) Explain NP-complete and NP-hard problem. (See Unit-V, Page 240, Prob.46)

8. Write short notes (any three) –
   (a) Graph traversal (See Unit-V, Page 231, Q.28)
   (b) Binary search tree (See Unit-V, Page 185, Q.1)
   (c) Parallel algorithm (See Unit-IV, Page 182, Q.33)
   (d) Branch and bound (See Unit-IV, Page 157, Q.23)
   (e) Minimum spanning tree. (See Unit-II, Page 71, Q.10)

RGPV

CS-4004 (CBGS)
B.E. (IV Semester) EXAMINATION, May 2018
Choice Based Grading System (CBGS)
ANALYSIS & DESIGN OF ALGORITHM

Note : (i) Attempt any five questions.

(ii) All questions carry equal marks.

1. (a) Explain the various criteria used for analyzing algorithm. (See Unit-I, Page 5, Q.6)

   (b) Write the merge sort algorithm and discuss its efficiency. Sort list E, X, A, M, P, L, E in alphabetical order using merge sort. (See Unit-I, Page 40, Q.47)

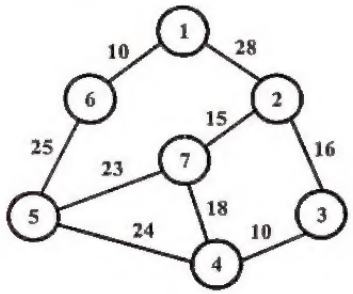
2. What is knapsack problem in greedy strategy ? Write the running time and recurrence relation of knapsack algorithm ? (See Unit-II, Page 77, Q.17)

3. Explain how to implement Warshall's algorithm without using extra memory for storing elements of the algorithms intermediate matrices. (See Unit-III, Page 124, Q.21)

4. Apply and explain the backtracking method to solve the following –
   (a) Hamiltonian circuit problem
   (b) Subset-sort problem. (See Unit-IV, Page 150, Q.17)

5. (a) Implement an algorithm for binary search. Discuss in detail about time complexity of binary search algorithm. (See Unit-I, Page 36, Q.39)

   (b) Apply Prim's algorithm to the following graph. Write their complexity. Find the minimum cost.



(See Unit-II, Page 92, Prob.8)

6. (a) Discuss in detail NP complete problems with example. (See Unit-V, Page 239, Q.45)

   (b) What is height balanced trees ? Why to be balanced a height in a trees ? (See Unit-V, Page 191, Q.6)

7. (a) Apply the branch and bound algorithms to solve the travelling salesman problem. Use suitable graph. (See Unit-IV, Page 162, Q.28)

   (b) Compare BFS and DFS. (See Unit-V, Page 231, Q.29)

8. Write a short notes (any four) –
   (a) B Trees (See Unit-V, Page 201, Q.13)
   (b) Parallel algorithm (See Unit-IV, Page 182, Q.33)
   (c) Multistage graphs (See Unit-III, Page 117, Q.15)
   (d) Quick sort (See Unit-I, Page 44, Q.53)
   (e) Graph coloring. (See Unit-IV, Page 151, Q.18)

RGPV

CE-4004 (CBGS)
B.E., IV Semester
EXAMINATION, November 2018
Choice Based Grading System (CBGS)
ANALYSIS AND DESIGN OF ALGORITHM

Note : (i) Attempt any five questions.
(ii) All questions carry equal marks.

1. (a) Briefly describe the asymptotic notations used in the study of algorithms. Also write the relational properties of the asymptotic notations. (See Unit-I, Page 14, Q.20)

   (b) Illustrate the operation of heapsort on the array –
   A = [5, 13, 2, 25, 7, 17, 20, 8, 4] (See Unit-I, Page 30, Prob.5)

2. (a) Write the algorithm for binary search. Apply binary search to find 122 in a list –
   44, 95, 104, 120, 144, 191, 198, 204, 244, 274, 122, 849, 904.
   (See Unit-I, Page 55, Prob.8)

   (b) Discuss the procedure for mergesort. Illustrate how the worst case performance of merge sort is better than the quicksort. Sort the given array of 8 elements step-by-step merge sort –
   A = [36, 19, 8, 13, 6, 24, 17, 4]. (See Unit-I, Page 58, Prob.10)

3. (a) Explain the concept behind greedy strategy. Using optimal merge pattern greedy method, merge the following files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.
   (See Unit-II, Page 67, Q.6)

   (b) A Knapsack capacity is W = 60. The weights and profits of four items are as follows –

Item	A	B	C	D
Profit P_i	280	100	120	120
Weight W_i	40	10	20	24

   Solve the Knapsack problem using Greedy strategy and find the maximum profit that can be obtained. (See Unit-II, Page 103, Prob.18)

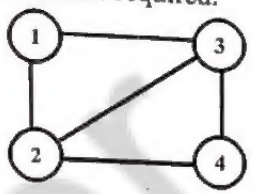
4. (a) There are 5 jobs whose profits $(P_1, P_2, P_3, P_4, P_5) = (60, 100, 20, 40, 20)$ and deadlines $(D_1, D_2, D_3, D_4, D_5) = (2, 1, 3, 2, 1)$. Find the optimal solution that maximizes profit on scheduling these jobs.
   (See Unit-II, Page 106, Prob.21)

   (b) Briefly explain the concept of dynamic programming. Discuss how a reliability design can be obtained using dynamic programming.
   (See Unit-III, Page 122, Q.18)

5. (a) Find the optimal solution for 0/1 Knapsack problem $(W_1, W_2, W_3, W_4) = (10, 15, 6, 9)$, $(P_1, P_2, P_3, P_4) = (2, 5, 8, 1)$ and W = 30.
   (See Unit-III, Page 129, Prob.4)

   (b) Explain 8 queen's problem and apply backtracking to solve this problem. (See Unit-IV, Page 143, Q.10)

6. (a) Colour the following graph using a vertex colouring algorithm. What is the minimum colours required.



(See Unit-IV, Page 157, Prob.2)

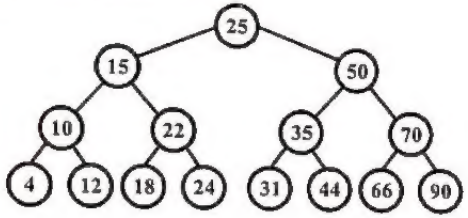
   (b) Consider the following initial cost matrix for Traveling Salesman Problem. Solve this problem using branch and bound technique –

$$\begin{bmatrix} \infty & 4 & 5 \\ 7 & \infty & 5 \\ 4 & 6 & \infty \end{bmatrix}$$

(See Unit-IV, Page 176, Prob.5)

7. (a) (i) Insert the following sequence of elements into an AVL tree, starting with an empty tree – 10, 20, 15, 25, 30, 16, 18, 19.
   (ii) Delete 30 in the AVL tree that you got and re-balance the tree (if required). (See Unit-V, Page 212, Prob.6)

   (b) Illustrate the inorder, preorder and postorder traversals for the following binary search tree.



(See Unit-V, Page 229, Q.26)

8. Write short notes on any two –
   (a) Strassen's matrix multiplication (See Unit-I, Page 51, Q.61)
   (b) Parallel algorithms (See Unit-IV, Page 182, Q.33)
   (c) B-tree (See Unit-V, Page 201, Q.13)
   (d) NP-Completeness. (See Unit-V, Page 239, Q.45)

RGPV

CE-402 (CBGS)
B.E., IV Semester
EXAMINATION, May 2019
Choice Based Grading System (CBGS)
ANALYSIS AND DESIGN OF ALGORITHM

Note : (i) Attempt any five questions.
(ii) All questions carry equal marks.
(iii) In case of any doubt or dispute the English version questio should be treated as final.

1. (a) What is an asymptotic notations ? Give the different notations to used to represent the complexity of algorithms.
(See Unit-I, Page 11, Q.15)

   (b) Give the divide and conquer solution for binary search and analyze its complexity.
(See Unit-I, Page 36, Q.41)

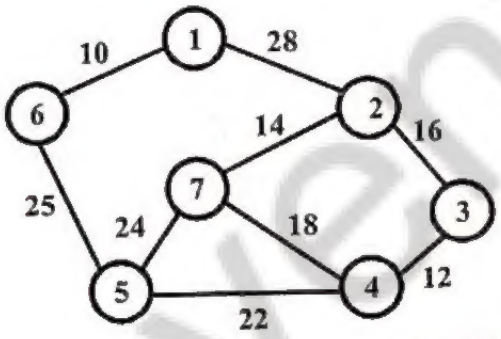
2. (a) Apply quicksort to sort the following list –
36, 12, 85, 79, 46, 18, 92, 30, 28, 65, 72. (See Unit-I, Page 60, Prob.14)

   (b) How can we prove that Strassen's matrix multiplication is advantageous over ordinary matrix multiplication ?
(See Unit-I, Page 53, Q.62)

3. (a) What is Knapsack problem ? How can we solve using Greedy approach ?
(See Unit-II, Page 76, Q.16)

   (b) Write algorithms for single source shortest path and find its complexity ?
(See Unit-II, Page 82, Q.24)

4. (a) Apply Kruskal's and Prim's algorithm for the following graph ? Write their time complexities. Find the minimum cost in each case.



(See Unit-II, Page 97, Prob.12)

   (b) What is multistage graph ? Write down its properties.
(See Unit-III, Page 117, Q.15)

5. (a) Write down the pseudocode for Floyd Warshall algorithm. Take one graph and apply this algorithm to find all pair shortest path on it.
(See Unit-III, Page 124, Q.21)

   (b) Write the recursive equation for 0/1 Knapsack problem based on the principles of optimality. Explain its execution strategy.
(See Unit-III, Page 117, Q.14)

6. (a) Write a detailed note on Parallel Algorithms.
(See Unit-IV, Page 182, Q.33)

   (b) What is backtracking ? Explain 8 queen's problem and how can we solve it using backtracking ?
(See Unit-IV, Page 143, Q.10)

7. (a) What is the meaning of lower bound theory and how can it be used in solving algebraic problems ?
(See Unit-IV, Page 180, Q.32)

   (b) What are B-trees ? Write down its properties ? What is the need for B tree ? What is the height of a B tree of order m ?
(See Unit-V, Page 202, Q.17)

8. Write short notes –
   (i) NP-completeness (See Unit-V, Page 239, Q.45)
   (ii) Tree Traversals (See Unit-V, Page 224, Q.21)
   (iii) Hamiltonian cycle (See Unit-IV, Page 148, Q.15)
   (iv) Graph coloring problem. (See Unit-IV, Page 151, Q.18)